ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації
(повне найменування інституту, назва факультету)
Кафедра прикладної математики та інформатики
(повна назва кафедри)

«До захисту допущено»
В.о. завідувача кафедри

_____ Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)
"___"_____20__ р.

Випускна кваліфікаційна робота

бакалавра
(освітній ступінь)

на тему: «Алгоритми оптимізації та стиснення даних»
зі спецчастиною: «Розробка та реалізація алгоритму Хаффмана з застосуванням C#»

Виконав (-ла): студент (-ка) 4 курсу, групи ___ІПЗ-19___
(шифр групи)

спеціальності 121 «Інженерія програмного забезпечення»
(код і назва спеціальності)

освітньої програми ___121 «Інженерія програмного забезпечення»__

_____Стреляєв Данило Андрійович_____ _____
(прізвище та ініціали) (підпис)

Керівник доц. каф. ПМІ, к.т.н., доцент Наталія МАСЛОВА _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Рецензент доц. каф. ЕТ, к.т.н., доцент Олександр ШТЕПА _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань

Студент _____
(підпис)

Луцьк – 2023р.

ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики
Освітній ступень бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(шифр і назва)
Освітня програма 121 Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)
«____»_____2023 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Стреляєву Данилу Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Алгоритми оптимізації та стиснення даних» зі спецчастиною «Розробка та реалізація алгоритму Хаффмана з застосуванням C#»
керівник роботи Маслова Н. О., к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом від « « 5» травня 2023 року № 175
2. Строк подання студентом роботи 08 червня 2023 року
3. Вихідні дані до роботи результати переддипломної практики

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) аналіз алгоритмів стиснення даних;
2) опис обґрунтування вибору алгоритму Хаффмана, як самого оптимального;
опис процесу роботи алгоритму Хаффмана;
3) опис процесу розробку поставленого завдання за допомогою C#;
проєктування поставленого завдання з використанням сучасних інструментів;
4) опис інструментів, що було використано під час виконання поставленого завдання;
5) тестування програмної розробки.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
робота містить 30 рисунків, 9 діаграм, 13 слайдів презентації та інший графічний матеріал, у кількості, достатній для демонстрації сутності роботи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Назарова І.А. доцент каф. ПМІ		7.06.2023

7. Дата видачі завдання 5 травня 2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
	Теоретичне опрацювання можливих алгоритмів стиснення та оптимізації даних	05.05.2023 – 10.05.2023	
	Теоретичне опрацювання алгоритму кодування Хаффманом	11.05.2023 – 15.05.2023	
	Визначення елементів розроблюємого програмного додатку	16.05.2023 – 20.05.2023	
	Створення програмного додатку	21.05.2023 – 01.06.2023	
	Тестування та пост-релізні зміни у програмному додатку	01.06.2023 – 07.06.2023	
	Підготовка пояснювальної записки	08.06.2023- 10.06.2023	
	Нормоконтроль та перевірка антиплагіатною системою пояснювальної записки	11.06.2023- 21.06.2023	
	Захист роботи	22.06.2023	

Студент _____ Данило СТРЕЛЯЄВ
(підпис) (Ім'я та ПРІЗВИЩЕ)

Керівник роботи _____ Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)

А Н О Т А Ц І Я

Стреляєв Данило Андрійович. Алгоритм оптимізації та стиснення даних/ Випускна кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» за спеціальністю 121 «Інженерія програмного забезпечення». ДВНЗ ДонНТУ, Луцьк, 2023.

Об’єкт дослідження – технології стиснення та оптимізації даних. Предмет дослідження – методи стиснення даних за допомогою методу Хаффмана.

Метою роботи є дослідження та використання методів побудови програмного додатку. Розробка програмного додатку, що буде ефективно реалізовувати роботу алгоритму шифрування та дешифрування Хаффмана, з використанням лінійного методу зберігання ключів дешифрування.

Методи досліджень базуються на сучасних положеннях стиснення даних, теорії шифрування, інтелектуального аналізу даних, сучасних застосуваннях в оптимізації та стиснення даних.

Наукова новизна полягає в створенні нестандартного методу зберігання ключа для дешифрування закодованого тексту алгоритмом Хаффмана.

Практичне значення полягає у створенні алгоритму, за допомогою якого можливе ефективне стиснення даних та можливість декодування зашифрованих даних зі стовідсотковою ймовірністю, без втрати даних.

Ключові слова: код Хаффмана, C#, VCS, GitHub, QA, test-cases.

Список публікацій здобувача

1. Стреляєв Д. Дослідження продуктивності web-серверу // Д.А. Стреляєв, В.І. Костін, вид.: І міжнародна науково-практична конференція студентів та молодих вчених «Математика та математичне моделювання у сучасному технічному університеті». Луцьк: Донецький національний технічний університет, 2022. С. 33 - 36.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

VCS – VERSION CONTROL SYSTEM

C# – C SHARP PROGRAMMING LANGUAGE

QA – QUALITY ASSURANCE

RLE – RUN LENGTH ENCODING

KWE – KEYWORD ENCODING

LZW – LEMPEL-ZIV-WELCH

ASCII – AMERICAN STANDART CODE FOR INFORMATION INTERCHANGE

ВСТУП

За увесь час існування сучасної «ідеї» розробки програмного забезпечення, існувало два основних параметри для класифікації програмних додатків: швидкість роботи програмного додатку та його використання ресурсів (будь то пам’яті або розрахункових можливостей), але з подальшим розповсюдженням все більш потужних пристроїв для розрахунків серед звичайних споживачів та корпорацій різного масштабу, з’явилась необхідність у шифрування даних через проблеми безпеки та конфіденційності. Так само з’явилась і необхідність у зменшенні об’ємів пам’яті, що використовуються на фоні сучасних технологій штучного інтелекту та обробки великих масивів даних. Для вирішення цих проблем може бути використаний алгоритм Хаффмана, який є ефективним оптимізатором об’єму даних та їх шифрувальником.

Мета роботи полягає у розробці програмного додатку, здатного шифрувати та дешифрувати будь який текст за допомогою алгоритму Хаффмана зі створенням ключей для шифрованих текстів.

Об’єкт дослідження – алгоритми оптимізації та стиснення даних, Спецчастина.

Предметом дослідження є методи розробки та реалізації алгоритму Хаффмана з застосуванням C#.

Практична реалізація результатів роботи - буде розроблено програмний додаток з використанням мови C# на базі ядра .NET, з використанням технологій систем контролю версій та тестування програмного додатку відносно методологій QA. Результатом дипломної роботи буде готовий програмний додаток, за допомогою якого, будь-який середньостатистичний користувач зможе виконати дії шифрування та дешифрування з використання алгоритму Хаффмана.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ АЛГОРИТМІВ ОПТИМІЗАЦІЇ ТА СТИСНЕННЯ ДАНИХ

1.1 Аналіз алгоритмів оптимізації та стиснення даних

Алгоритми стиснення даних діляться на дві основні групи:

- Стиснення без втрат інформації;
- Стиснення із втратами інформації.

Стиснення без втрат інформації використовується при необхідності «архівації» даних для зменшення об’єму даних без спотворення кінцевих даних, після усіх перетворень.

Ця група алгоритмів використовується для такої інформації як:

- текст;
- аудіо-, фото- та відео- файли, що не повинні спотворюватися при стисненні;
- текст, що містять дані невідомого формату

Алгоритми стиснення без втрат можуть бути використанні для будь-якої інформації, але, порівнюючи їх з стисненням із втратами, об’єм інформації, що зменшується, буде більшим. Для максимального стиснення, можливі поєднання декількох методів стиснення з та без втрат.

Наприклад, при стисненні фотографії, можливі графічні перетворення зі звуженням початкового фото, де кожні дев’ять пікселів усереднюються, та перетворюються у нову графічну одиницю та займають місце попередніх дев’яти пікселів. Це перетворення є алгоритмом стиснення для фотографії, що зменшує об’єм фото, відносно параметрів стиснення.

Після того, як було отримано зменшене, за об’ємом, зображення, може бути використаний будь-який інший алгоритм стиснення, наприклад, алгоритм Хаффмана або алгоритм кодування довжини серій.

Після цих перетворень, об’єм перетворених даних, відносно початкових, може бути, у деяких випадках, набагато меншим, аніж при використанні будь-

якого типу стиснення по-окремості.

Типи алгоритмів стиснення без втрат діляться на три різні групи:

- Ентропійний тип;
- Словниковий тип;
- Інші типи (кодування без використання ентропійного та словникового кодування).

Усі ці типи розділяють алгоритми за головною ідеєю, що є основою самих алгоритмів.

Ентропійний тип, або ентропійне кодування – це кодування, яке намагається усереднити ймовірність появи елементів текстової послідовності (символів та послідовностей символів) у закодованій послідовності. Алгоритми цього типу кодування намагаються створити словник для перетворень таким чином, щоб найчастіші символи або послідовності символів мали найменшу довжину – чим більше інтенсивність символу у тексті, тем менша його кодована довжина.

Прикладами алгоритмів ентропійного типу є:

- Алгоритм Хаффмана;
- Алгоритм Шеннона;
- Коди Голомба.

Словниковий тип майже не відрізняється від ентропійного типу, але у цьому типі, увага на інтенсивність появи символів не звертається. Принцип додавання символів до словнику відповідає алгоритму роботи стеку – перший символ, що оброблюється, додається в словник, усі наступні невідомі символи, при їх обробці, додаються у словник за попереднім символом. Усі алгоритми словникового типу є варіаціями та модифікаціями алгоритму RLE. Сам принцип словникового кодування є узагальненою ідеєю алгоритму RLE.

Прикладами алгоритмів словникового типу є:

- Deflate;
- Snappy;
- LZ77/LZ78 (Та їх модифікації LZW, LZMA та інші).

Інші типи стиснення, частіш за все, є допоміжними алгоритмами для перетворення даних у більш оптимізований вид для подальших стиснень даних, або такі методики стиснення даних, що не можна класифікувати за будь яким іншим типом.

Прикладами алгоритмів словникового типу є:

- RLE;
- Delta-кодування;
- Перетворення Берроуза-Вілера.

Основними прикладами алгоритмів стиснення даних без втрат інформації є:

- Алгоритм кодування довжин серій (RLE);
- Алгоритм Хаффмана;
- Алгоритми словникового методу (KWE).

Усі ці алгоритми мають один концепт – перетворення повторюваних елементів у більш оптимізовану послідовність даних з використанням словників (окрім алгоритму кодування довжин серій).

Алгоритм кодування довжин серій є доволі простим, у розумінні та реалізації, алгоритмом стиснення.

Основною ідеєю цього алгоритму є пошук символів, що повторюються один за одним, після чого ця послідовність символів перетворюється у одиницю кодування RLE у вигляді: «кількість символів, що повторюються | символ».

Наприклад, є текстова послідовність «АААВВВВВАБВАВБА». Довжина початкового тексту складає 15 символів. Якщо стиснути цю послідовність за допомогою алгоритму кодування довжини серій, то буде отримано таку стиснену версію початкового тексту: «3А5В1А1Б1В1А1В1Б1А».

Довжина стисненого тексту складає 18 символи. Так як перетворена інформація є символьною та не є бітовою, то це перетворення не є ефективним.

Для пошуку коефіцієнту ефективності стиснення необхідно використати

відповідну формулу.

Формула розраховується за допомогою ділення кінцевої довжини тексту на початкову. Якщо коефіцієнт дорівнює одиниці, то стиснення не відбулось. Якщо коефіцієнт більше одиниці, то відбулося збільшення об’єму даних. При коефіцієнті менше одиниці відбувається стиснення даних.

Розрахунок ефективності стиснення розраховується за формулою:

$$k_s = \frac{d_s}{d_p},$$

де

k_s – коефіцієнт стиснення даних;

d_s – довжина стисненого тексту;

d_p – довжина початкового тексту.

Якщо розрахувати коефіцієнт ефективності стиснення для попереднього прикладу, то буде отримано наступні результати:

$$k_s = \frac{18}{15} = 1{,}2.$$

З наведених розрахунків, можна визначити, що довжина тексту після перетворення збільшилася на 20%. Основною проблемою є символи, що повторюються тільки один раз, через що один символ перетворюється на два, та кількість символів що повторюються один раз є набагато більша, аніж кількість символів що повторюються. Також, при послідовності символів з довжиною два також не відбувається стиснення, через однакову кількість початкових та кінцевих символів.

Проблему послідовностей одиничних символів можна вирішити за допомогою від’ємних коефіцієнтів, які вказують на кількість символів, що йдуть послідовно без повторень. При використанні цього правила при перетворені, попередній результат буде мати такий вигляд: «3А5В-7АБВАВБА».

Після перетворення, довжина перетвореного тексту дорівнює 13 символів. Коефіцієнт стиснення дорівнює приблизно 0,86, тому ефективність стиснення тексту дорівнює 13%.

Найбільша ефективність цього алгоритму відображається при його використанні з вхідними даними, що мають велику кількість повторюваних символів у ланцюгах, наприклад, бінарних даних (якщо це є доцільним, відносно перетворення даних з бітового формату даних до рядкового) та простих графічних зображень. При роботі алгоритму RLE також не з’являється необхідності у створенні ключів, тому що вихідний текст, фактично, більше оптимізується, аніж перетворюється, бо формат тексту та його тип даних залишається таким самим.

Алгоритм кодування Хаффманом є типом ентропійного кодування, головною метою якого є генерація словника для перетворення вхідних даних таким чином, щоб для шифрування та дешифрування можна було використати розроблений ключ - бінарне дерево Хаффмана.

На відміну від інших методів оптимізації – алгоритм кодування Хаффманом є завжди оптимальним.

Алгоритм кодування Хаффманом ділиться на два етапи:

- побудова оптимального бінарного дерева Хаффмана, відносно вхідного тексту;
- кодування вхідного тексту за допомогою створеного дерева Хаффмана.

Основною ідеєю за цим методом кодування є перетворення усіх символів вхідного тексту у бінарні еквіваленти таким чином, щоб найчастіші символи (що мають найбільшу кількість повторень у тексті) мали найкоротше кодування, а символи з найменшою інтенсивністю – найбільшу довжину кодування. Таким чином йде усереднення кількості символів у тексті, так як велика довжина символів з маленькою інтенсивністю компенсується маленькою довжиною символів з великою інтенсивністю.

Для прикладу роботи цього алгоритму, було узято текст «Інженерія програмного забезпечення».

Спочатку, необхідно побудувати основу дерева Хаффмана, тобто таблицю частот, яка розбиває текст на індивідуальні символи та шукає їх

інтенсивність у тексті. Але перед виконанням цього кроку, необхідно визначитися з правилами класифікації символів. Для цього прикладу, будуть використовуватися такі правила:

- прописні та маленькі символи не однакові;
- пробіли у тексті вважаються за символи.

Після визначення правил, можна починати роботу над створенням таблиці частот.

Розроблену таблицю для обраного тексту зображено на рисунку 1.1.

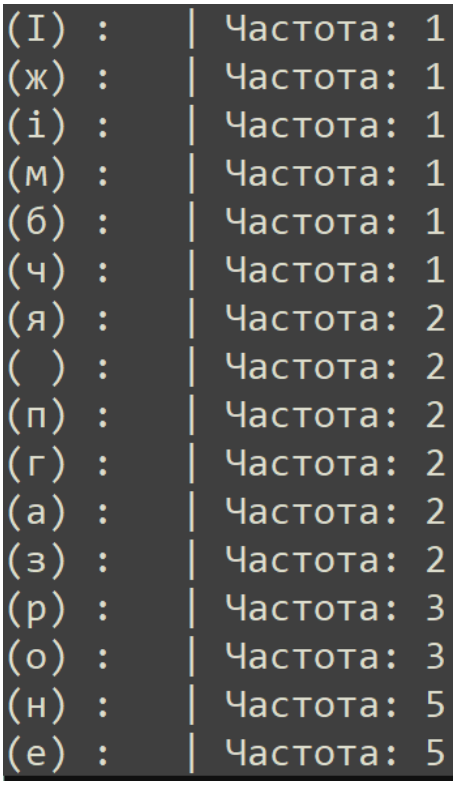


Рисунок 1.1 – Таблиця частот символів у вхідному масиві даних

Після створення таблиці частот, наступним кроком є побудова дерева Хаффмана, використовуючи розроблену таблицю як основу. Кінцевим результатом цього дерева буде корінь – значення якого буде відповідати сумі усіх частот символів.

Алгоритм побудови дерева Хаффмана має наступні кроки:

- знайти у дереві два листу з найменшою частотою;
- поєднати їх у нову точку, значення якої буде відповідати сумі

частот об’єднаних листу;

- якщо кількість точок дорівнює одному - закінчити алгоритм, у іншому випадку, почати алгоритм з початку.

Якщо діяти відносно наведеного алгоритму, то можна побудувати дерево для обробляємого тексту. Результат роботи алгоритму побудови дерева Хаффмана зображено на рисунку 1.2.

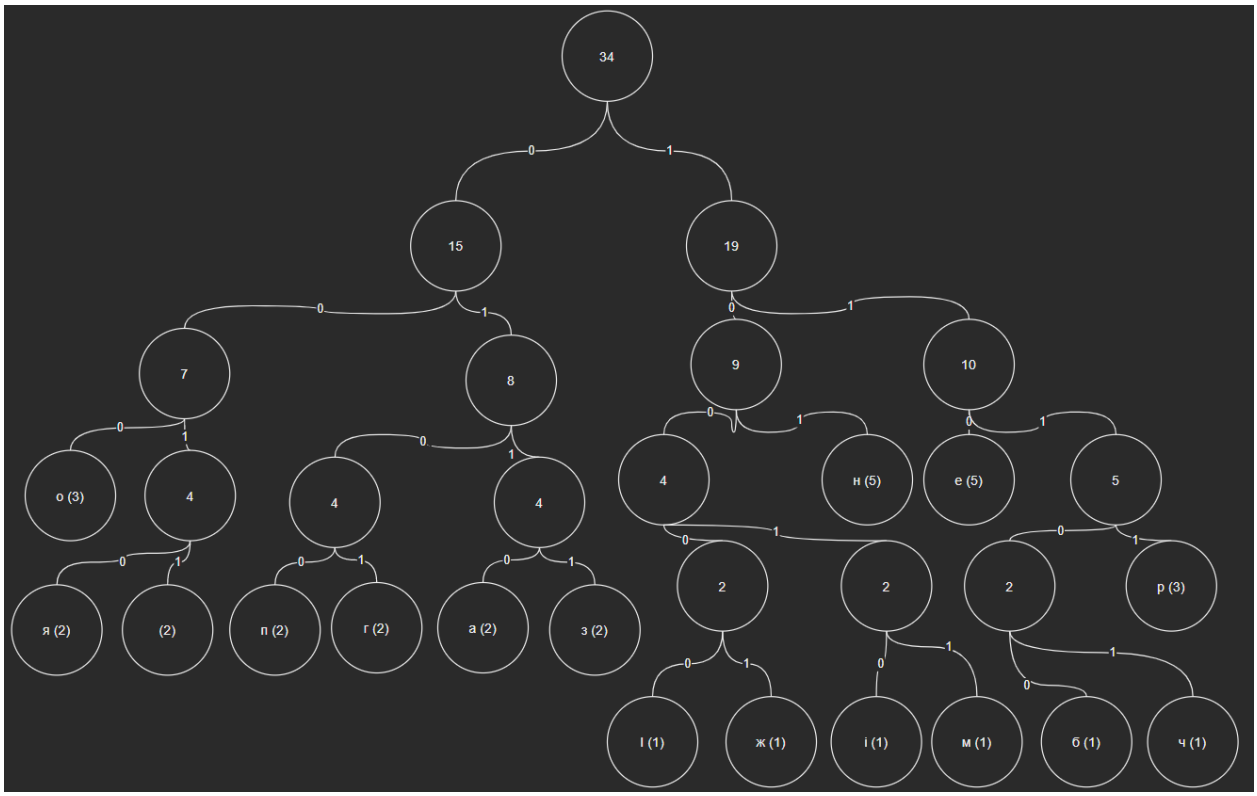


Рисунок 1.2 – Побудоване дерево Хаффмана

На побудованому дереві можна побачити, що символи з високою частотою знаходяться ближче до кореня, через що їх код буде значно меншим за коди символів, що знаходяться якнайдовше від кореня дерева.

За допомогою створеного дерева можливе кодування вхідного тексту. Алгоритм кодування полягає у перетворені символів до з рядкового типу даних (або будь-якого іншого типу даних) до бінарного за допомогою проходження по листах дерева та «збирання» коду з пройденого шляху. У цьому дереві лівий лист має бінарне значення нуля, правий одиниці.

Таким чином, для кодування символу «І» буде пройдений шлях «19>9>4>2>І», що транслюється у бінарні значення пройденого шляху «1>0>0>0>0». Виходячи з цього, закодоване значення символу «І» буде відповідати бінарна послідовність «10000».

Якщо узяти більший, за частотою, символ, наприклад «н», частота якого дорівнює п’яти, ми отримаємо шлях «19>9>н», що переводиться у шлях «101».

Обробивши усі інші символи, було отримано закодований текст, який має наступний вигляд:

«10000101100011101011101111100100010001101001111000010111110110100111010000101000001101110110111001100111010011011101110101101001 0».

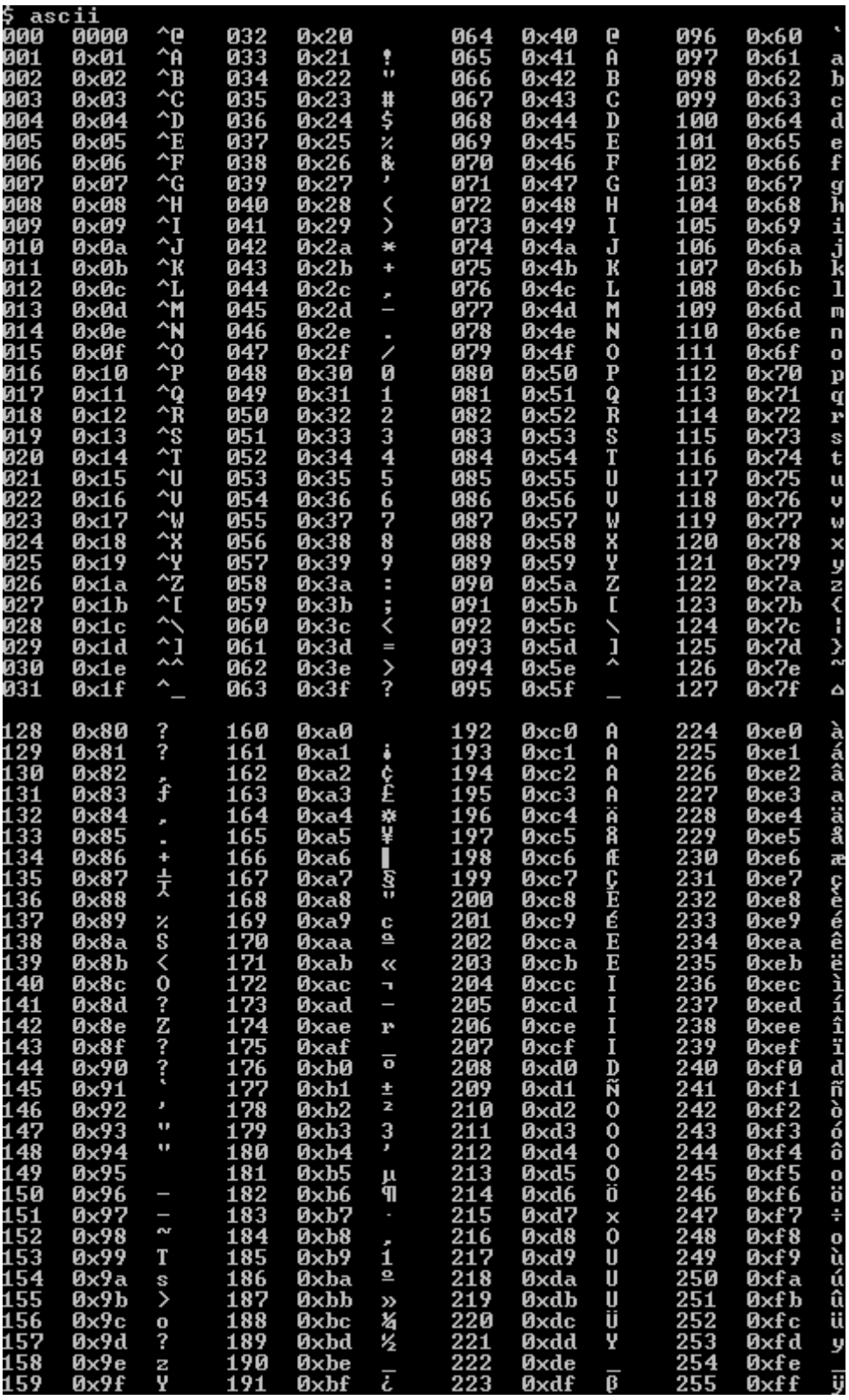
Для оцінки коефіцієнту стиснення, переведемо дані текстового формату у бітовий (за системою ASCII), та підрахуємо кількість зайнятої пам’яті для оригінального тексту та закодованого. Кількість біт, які зайняті у пам’яті при переведенні тексту до бінарного типу складає 305 біт. Кількість біт для закодованого тексту складає 129 біт. Коефіцієнт стиснення складає 0,42, тобто оригінальний текст був стиснений на 58%. Ці результати показують, що кодування Хаффманом є ефективним інструментом для стиснення даних (у цьому випадку – тексту).

Якщо реалістично оцінювати ефективність стиснення саме у цьому прикладі – воно не ефективне через необхідність зберігання великого ключу для дешифрування, який займає, відносно оригінального тексту, великий об’єм пам’яті, тому цей алгоритм є більш доцільним при кодуванні великих об’ємів даних, де алгоритм буде зберігати приблизно той же рівень ефективності стиснення у більшості випадків, через що об’єми звільняємої пам’яті будуть дуже значними.

Алгоритми словникового типу, здебільшого, є варіаціями алгоритму кодування «LZ77/LZ78» – кодування Лемпеля-Зіва. Назва створена із фамілій авторів та року створення алгоритму. Цей метод не є актуальним, та частіше використовуються його модифікації, як, наприклад, LZW – кодування Лемпеля-Зіва-Велча.

Суть методу полягає у алгоритмі, під час роботи якого, відносно вхідних даних у вигляді тексту, буде створюватися словник, який буде перетворювати вхідний набір символів або символ у іншу форму, частіш за все – бінарні рядки, але можливі використання і інших систем числення та репрезентації даних. Частіш за все словник має набір початкових даних, що зберігає у собі перші 256 символів системи ASCII, які є основою для подальшого додавання нових символів (або рядків символів) до словника, створюючи нові записи із вже існуючих записів, для подальшого зменшення об’єму тексту, або додавання нового елементу до словника.

Для прикладу роботи цього алгоритму, було узято вхідний текстовий рядок «Software/Software». Для основи словника буде узято збільшену таблицю ASCII, яка зображена на рисунку 1.3. Таблиця має три стовпців, де будуть використовуватися перший та останній стовпець, де перший є індексом запису у словнику, як перетворення для кодування, а останній відображає символ запису словника.



$ ascii

000	0000	^@	032	0x20		064	0x40	@	096	0x60	`
001	0x01	^A	033	0x21	!	065	0x41	A	097	0x61	a
002	0x02	^B	034	0x22	"	066	0x42	B	098	0x62	b
003	0x03	^C	035	0x23	#	067	0x43	C	099	0x63	c
004	0x04	^D	036	0x24	$	068	0x44	D	100	0x64	d
005	0x05	^E	037	0x25	%	069	0x45	E	101	0x65	e
006	0x06	^F	038	0x26	&	070	0x46	F	102	0x66	f
007	0x07	^G	039	0x27	'	071	0x47	G	103	0x67	g
008	0x08	^H	040	0x28	(	072	0x48	H	104	0x68	h
009	0x09	^I	041	0x29	)	073	0x49	I	105	0x69	i
010	0x0a	^J	042	0x2a	*	074	0x4a	J	106	0x6a	j
011	0x0b	^K	043	0x2b	+	075	0x4b	K	107	0x6b	k
012	0x0c	^L	044	0x2c	,	076	0x4c	L	108	0x6c	l
013	0x0d	^M	045	0x2d	-	077	0x4d	M	109	0x6d	m
014	0x0e	^N	046	0x2e	.	078	0x4e	N	110	0x6e	n
015	0x0f	^O	047	0x2f	/	079	0x4f	O	111	0x6f	o
016	0x10	^P	048	0x30	0	080	0x50	P	112	0x70	p
017	0x11	^Q	049	0x31	1	081	0x51	Q	113	0x71	q
018	0x12	^R	050	0x32	2	082	0x52	R	114	0x72	r
019	0x13	^S	051	0x33	3	083	0x53	S	115	0x73	s
020	0x14	^T	052	0x34	4	084	0x54	T	116	0x74	t
021	0x15	^U	053	0x35	5	085	0x55	U	117	0x75	u
022	0x16	^V	054	0x36	6	086	0x56	V	118	0x76	v
023	0x17	^W	055	0x37	7	087	0x57	W	119	0x77	w
024	0x18	^X	056	0x38	8	088	0x58	X	120	0x78	x
025	0x19	^Y	057	0x39	9	089	0x59	Y	121	0x79	y
026	0x1a	^Z	058	0x3a	:	090	0x5a	Z	122	0x7a	z
027	0x1b	^[	059	0x3b	;	091	0x5b	[	123	0x7b	{
028	0x1c	^\	060	0x3c	<	092	0x5c	\	124	0x7c	|
029	0x1d	^]	061	0x3d	=	093	0x5d	]	125	0x7d	}
030	0x1e	^^	062	0x3e	>	094	0x5e	^	126	0x7e	~
031	0x1f	^_	063	0x3f	?	095	0x5f	_	127	0x7f	⌂
128	0x80	?	160	0xa0		192	0xc0	A	224	0xe0	à
129	0x81	?	161	0xa1	¡	193	0xc1	A	225	0xe1	á
130	0x82	,	162	0xa2	¢	194	0xc2	A	226	0xe2	â
131	0x83	ƒ	163	0xa3	£	195	0xc3	A	227	0xe3	a
132	0x84	„	164	0xa4	¤	196	0xc4	Ä	228	0xe4	ä
133	0x85	.	165	0xa5	¥	197	0xc5	Å	229	0xe5	å
134	0x86	+	166	0xa6	¦	198	0xc6	Æ	230	0xe6	æ
135	0x87	‡	167	0xa7	§	199	0xc7	Ç	231	0xe7	ç
136	0x88	^	168	0xa8	¨	200	0xc8	È	232	0xe8	è
137	0x89	‰	169	0xa9	©	201	0xc9	É	233	0xe9	é
138	0x8a	S	170	0xaa	ª	202	0xca	E	234	0xea	ê
139	0x8b	<	171	0xab	«	203	0xcb	E	235	0xeb	ë
140	0x8c	O	172	0xac	¬	204	0xcc	I	236	0xec	ì
141	0x8d	?	173	0xad	-	205	0xcd	I	237	0xed	í
142	0x8e	Z	174	0xae	®	206	0xce	I	238	0xee	î
143	0x8f	?	175	0xaf	¯	207	0xcf	I	239	0xef	ï
144	0x90	?	176	0xb0	°	208	0xd0	D	240	0xf0	d
145	0x91	‘	177	0xb1	±	209	0xd1	Ñ	241	0xf1	ñ
146	0x92	’	178	0xb2	²	210	0xd2	O	242	0xf2	ò
147	0x93	"	179	0xb3	3	211	0xd3	O	243	0xf3	ó
148	0x94	"	180	0xb4	´	212	0xd4	O	244	0xf4	ô
149	0x95	•	181	0xb5	µ	213	0xd5	O	245	0xf5	o
150	0x96	–	182	0xb6	¶	214	0xd6	Ö	246	0xf6	ö
151	0x97	—	183	0xb7	·	215	0xd7	x	247	0xf7	÷
152	0x98	~	184	0xb8	¸	216	0xd8	O	248	0xf8	o
153	0x99	T	185	0xb9	1	217	0xd9	U	249	0xf9	ù
154	0x9a	s	186	0xba	º	218	0xda	U	250	0xfa	ú
155	0x9b	>	187	0xbb	»	219	0xdb	U	251	0xfb	û
156	0x9c	o	188	0xbc	¼	220	0xdc	Ü	252	0xfc	ü
157	0x9d	?	189	0xbd	½	221	0xdd	Y	253	0xfd	y
158	0x9e	z	190	0xbe	¾	222	0xde	¯	254	0xfe	¯
159	0x9f	Y	191	0xbf	¿	223	0xdf	ß	255	0xff	ÿ

Рисунок 1.3 – Стандартна ASCII таблиця

Алгоритм кодування за допомогою алгоритму LZW складається з наступних кроків:

- прочитати поточний символ;

- якщо символ є у словнику, перейти до обробки наступного символу з додаванням попереднього символу (або символів), що оброблювалися на попередньому кроці, виконати цей крок ще раз, якщо символу немає у словнику перейти на наступний крок, якщо було досягнуто кінця тексту – перейти до останнього кроку;
- новий символ буде доданий до словника для подальшого використання, вихідним кодуванням для оброблюємого символу або символів буде попередній оброблюємий символ/и, що був у словнику до моменту створення нового запису, поточним символом стає останній символ, через який було створено новий запис у словнику, перейти до першого кроку;
- встановити необхідний код замість поточного символу та завершити роботу алгоритму.

На тексті прикладу можна побачити принцип роботи розробленого алгоритму. Першим поточним символом буде «S», якщо звіритися з таблицею словника, то такий запис є наявним, він має номер «083». Через те, що такий запис вже у словнику, то до поточного символу додається наступний, це символ «o», через що буде отримано набор символів «So». Перевіривши таблицю словника, можна побачити що такого набору символів нема, тому буде додано новий запис, що буде мати назву «So» та номер індексу «256». Вихідним значенням поточного етапу буде значення запису «S», тому вихідні дані, на поточний момент, матимуть вигляд «083». Далі, поточним символом стає останній символ, що викликав необхідність додавання нового запису до словника – символ «o» на другій позиції текстового рядка.

Виконавши однакові дії для символу «o», як при обробці першого символу «S», знов буде отримано новий запис «of», який матиме індекс «257», закодований вихідний рядок матиме вид «083 111», так як вихідне значення поточного символу дорівнює «111», наступним символом буде буква на третій позиції тексу – «f». Виконавши ці дії для наступних символів, які повністю повторювані за алгоритмом обробки з попередніми прикладами, поточний символ дійде до букви «S» на початку другого слова у текстовому рядку. При

перевірці цього символу, як і у першому випадку, до поточного символу буде доданий наступний, але символ «So» також вже є у словнику, тому до цього символу додається ще один символ «f», через що буде отримано символ «Sof», якого вже нема у словнику, тому для нього буде проведена стандартний алгоритм додавання запису у словник, а вихідне значення для поточного символа вже буде інакшим, а саме «083 111 … 256», а наступним поточним символом буде «f» на третій позиції другого слова у текстовому рядку.

Як можна побачити з наведеного прикладу, цей алгоритм має великий потенціал, але через великий об’єм словника – він може бути неефективним при обробці малих об’ємів даних, тому цей метод є найефективнішим при використанні його з великими даними.

Також, для цього кодування необхідне використання роздільників у кінцевому, закодованому, тексті, тому, як мінімум, один символ повинен бути зарезервований для розділення закодованих записів між собою.

Цей метод, у реалістичних умовах, повинен реалізовуватися з певними обмеженнями на довжину та кількість записів у словнику, так як без обмежень, кількість записів може бути безкінечною.

Ще, можна зауважити необхідність у повторній обробці вхідного тексту після перших обробок, тому, що доки у словник додаються нові записи при обробці – наступна ітерація стиснення може принести кращі результати за попередні, тому максимальна ефективність цього методу може потребувати більшу кількість часу, на відміну від альтернатив.

1.2 Висновки за розділом

У цьому розділі було розглянуто типи алгоритмів стиснення та оптимізації та їх методи, які використовуються на сьогоднішній день, або є основою багатьох інших алгоритмів. Було виявлено недоліки та позитивні риси кожного з опрацьованих алгоритмів.

2 АНАЛІЗ ЕФЕКТИВНОСТІ ОБРАНОГО МЕТОДУ ОПТИМІЗАЦІЇ

2.1 Аналіз ефективності кодування даних Хаффманом відносно інших методів стиснення.

Вхідні дані, що кодуються за допомогою алгоритмів стиснення та оптимізації, мають два параметри, відносно яких їх можна розділяти між собою:

- кількість індивідуальних символів;
- довжина тексту.

Ці параметри дозволяють оптимально налаштувати вхідні дані, для кожного з методів стиснення. Методи стиснення у прикладах з попереднього розділу були обрані для аналізу у цьому розділі, окрім методу RLE, так як його використання потребує певних наборів даних, які є специфічними та не розповсюдженими, як, наприклад, звичайний текст.

Для тестування ефективності алгоритмів, було обрано декілька наборів текстових даних, в яких використовується різні налаштування зазначених параметрів.

Тестові блоки для кожного з методів були розділені на 2 групи:

- перша група, де зберігається пропорційне зростання кількості індивідуальних символів та довжини тексту, симулюючи реалістичні умови використання алгоритму для стиснення тексту;
- друга група, де зберігається підвищення довжини тексту, але інтенсивність зростання кількості індивідуальних символів зменшена, що відображає менш ймовірний, але реалістичний сценарій.

Необхідно зауважити, що результати тестування не беруть до уваги розмір створеного словника, бо доцільність тестування при цьому може варіюватися від теста до тесту.

Спочатку було протестовано стиснення кодуванням Хаффмана. Діапазон довжини вхідного тексту варіюється від 5 до 5000 з використанням

дев’яти різних кроків для кожної із груп.

На рисунку 2.1 зображено графік порівнянням гістограм початкового та стиснутого розміру тексту, з графіком відсотку стиснення відносно довжини тексту. На рисунку 2.2 зображено графік кількості індивідуальних символів та відсотку стиснення для першої групи.

Рисунок 2.1 – Тестування стиснення Хаффманом, перша група з графіком відсотку стиснення

Рисунок 2.1 відображає стабільність роботи алгоритму стиснення Хаффманом при збільшенні кількості індивідуальних символів та довжини тексту.

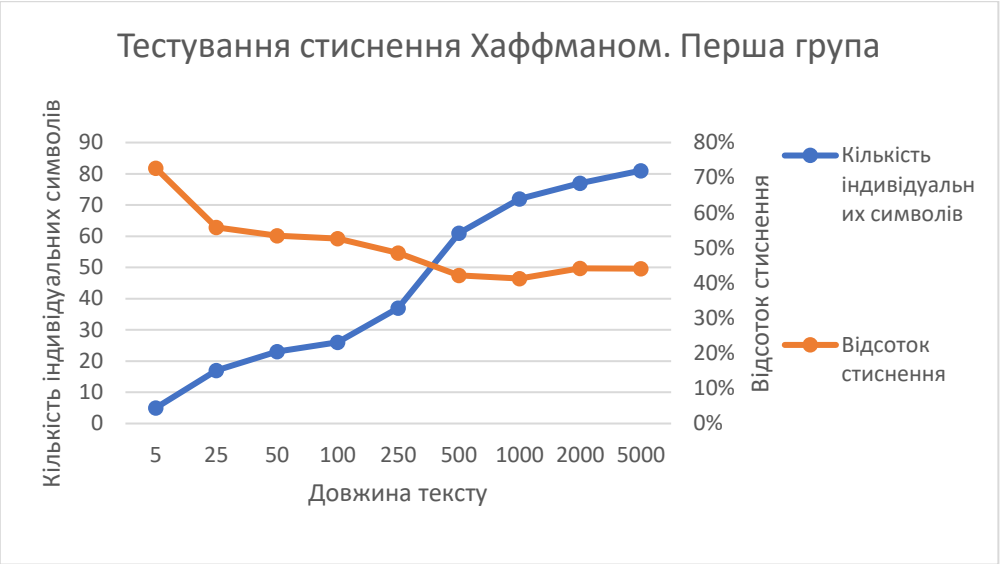


Рисунок 2.2 – Тестування стиснення Хаффманом, перша група з графіками відсотку стиснення та кількості індивідуальних символів

На перших ітераціях можна побачити (рисунок 2.2), що маленька кількість індивідуальних символів впливає на відсоток стиснення, та при збільшенні кількості символів – відсоток спадає, але така мала вибірка даних не дає можливості зробити точний висновок з цього приводу, саме тому необхідно зробити аналіз другої групи.

На рисунку 2.3 відображено такий самий графік, як і на рисунку 2.1, але у цьому випадку використовуються дані з групи два.

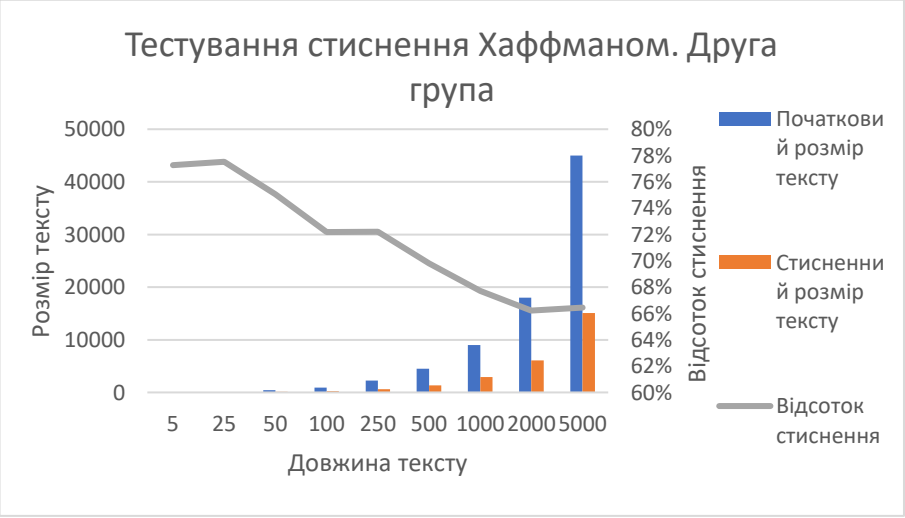


Рисунок 2.3 – Тестування стиснення Хаффманом, друга група з графіком відсотку стиснення

На рисунку 2.3 можна побачити зберігання великого відсотку стиснення на усіх етапах, але відсоток значно зменшується при збільшенні кількості індивідуальних символів, якщо порівнювати перший тест та останній.

На рисунку 2.4 зображено графік кількості індивідуальних символів та відсотку стиснення для другої групи.

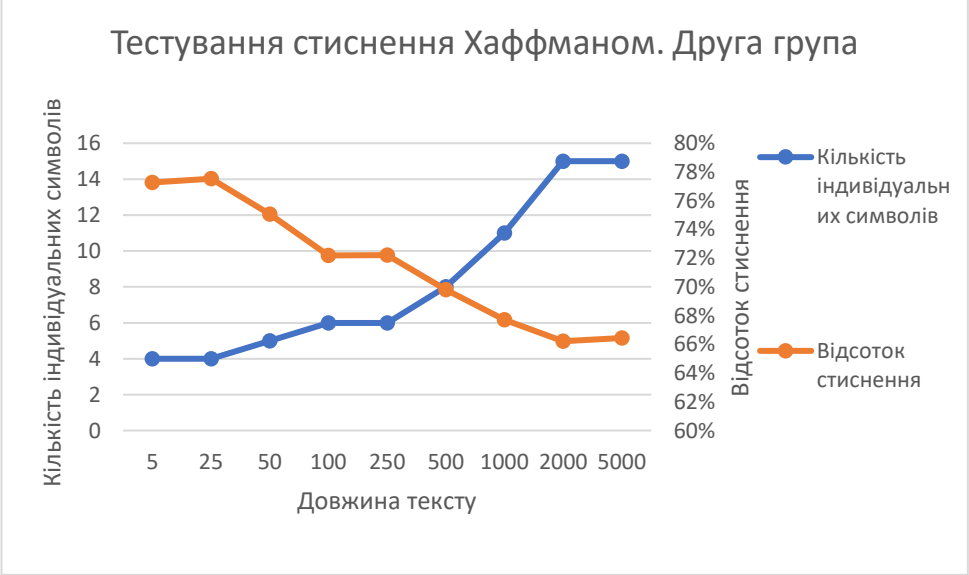


Рисунок 2.4 – Тестування стиснення Хаффманом, друга група з графіками відсотку стиснення та кількості індивідуальних символів

Рисунок 2.4 чітко відображає залежність кількості індивідуальних символів від відсотку стиснення. Загальна форма графіку першої групи відповідає другій, але при повільному та рівномірному зростанні кількості символів у тестах – графік відсотку стиснення є більш стабільним, чого не можна сказати про графік другої групи, де збільшення кількості символів з 6 до 15 зменшує відсоток стиснення з 72% до 66%.

Отримані дані є логічним результатом роботи алгоритму Хаффмана, тому що при меншій кількості індивідуальних символів – максимальна довжина закодованого символу зменшується, через що загальна кількість бінарних даних, що отримуються наприкінці роботи алгоритму значно менша. Довжина тексту не має великої ролі, тому що ефективність стиснення залишається майже однаковою тоді, коли кількість індивідуальних символів

залишається статичною.

Після аналізу результатів алгоритму Хаффмана, було проведено такі ж самі тести з використанням алгоритму LWZ, використовуючи однакові дані.

Треба зазначити, що у цих тестах, роздільного символу між закодованими даними не було. Вважається, що довжина кожного елементу відповідає максимально зазначеній довжині, яка зберігається у словнику, тому при декодуванні буде зчитуватися однакова кількість символів, через що необхідності у їх розділенні не виникне.

На рисунках 2.5 та 2.6 зображено графіки, аналогічні до графіків 2.1 та 2.2, відповідно, але побудовані на результатах тестування алгоритму LWZ.

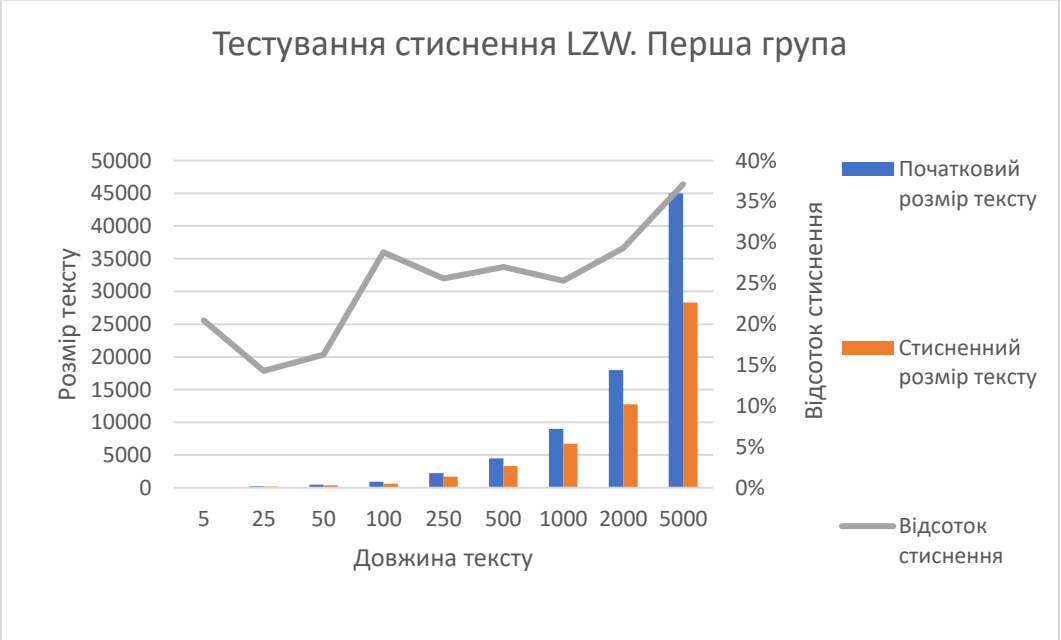


Рисунок 2.5 – Тестування стиснення LZW, перша група з графіком відсотку стиснення

Рисунок 2.5 відображає зростання значення відсотку стиснення при збільшенні довжини та кількості індивідуальних символів. Це є логічним результатом роботи цього алгоритму, бо чим довший початковий текст, тим більше доповнюється словник, через що текст буде максимально оптимізований, через створення достатньої кількості нових кодувань, які однією одиницею коду стискають більше і більше символів.

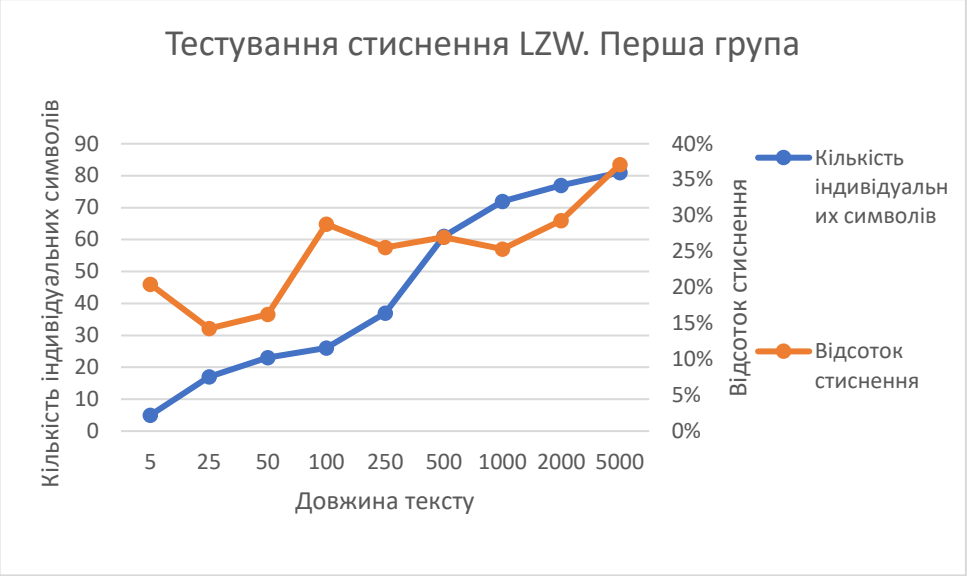


Рисунок 2.6 – Тестування стиснення LZW, перша група з графіками відсотку стиснення та кількості індивідуальних символів

Графік 2.6 ще більше відображає залежність довжини тексту від його відсотку стиснення.

Так само, як і для алгоритму Хаффмана, алгоритм LZW обробив другу групу даних. Рисунки 2.7 та 2.8 відповідають рисункам 2.3 та 2.4, але вони відображають дані, отримані при тестуванні за допомогою алгоритму LZW.

Рисунок 2.7 – Тестування стиснення LZW, друга група з графіком відсотку стиснення

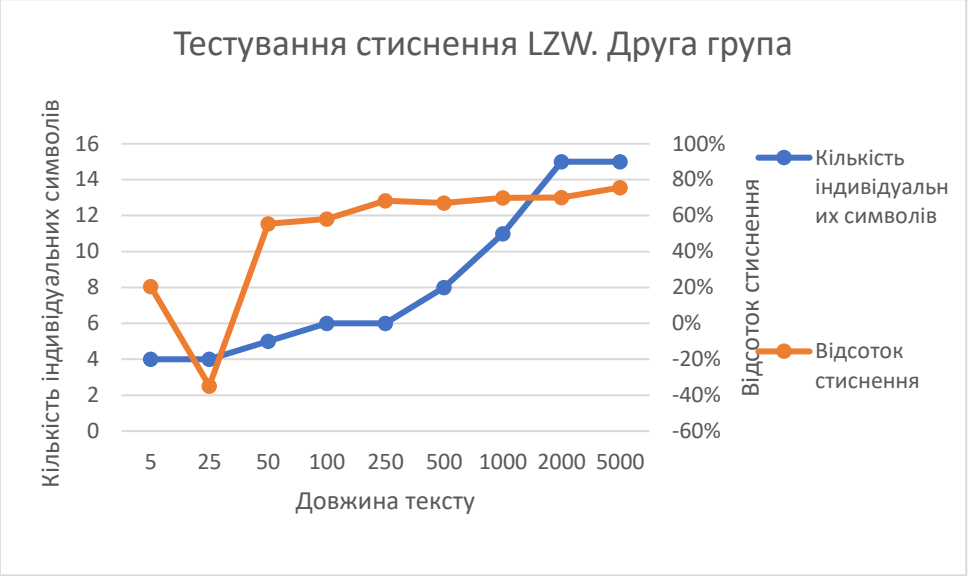


Рисунок 2.8 – Тестування стиснення LZW, друга група з графіками відсотку стиснення та кількості індивідуальних символів

На рисунка 2.7 та 2.8 можна побачити стабільний відсоток стиснення, як і його високе значення, на, майже, усіх етапах тестування, що вказує на залежність цього алгоритму від кількості індивідуальних символів, так само як і у алгоритмі Хаффмана.

На другому етапі тестування було виявлено неефективну роботу алгоритму, через що текст зазнав негативного стиснення і збільшився на 35%. Стабільного відтворення цієї «аномалії» не є можливим, тому це можна класифікувати, як випадкова помилка через наданий набір даних, але у той же час можна зазначити, що при використані алгоритму Хаффмана – таких проблем не було виявлено.

Для остаточного порівняння двох алгоритмів, на рисунку 2.9 було зображено графік порівняння ефективності стиснення для кожного алгоритму кожної групи.

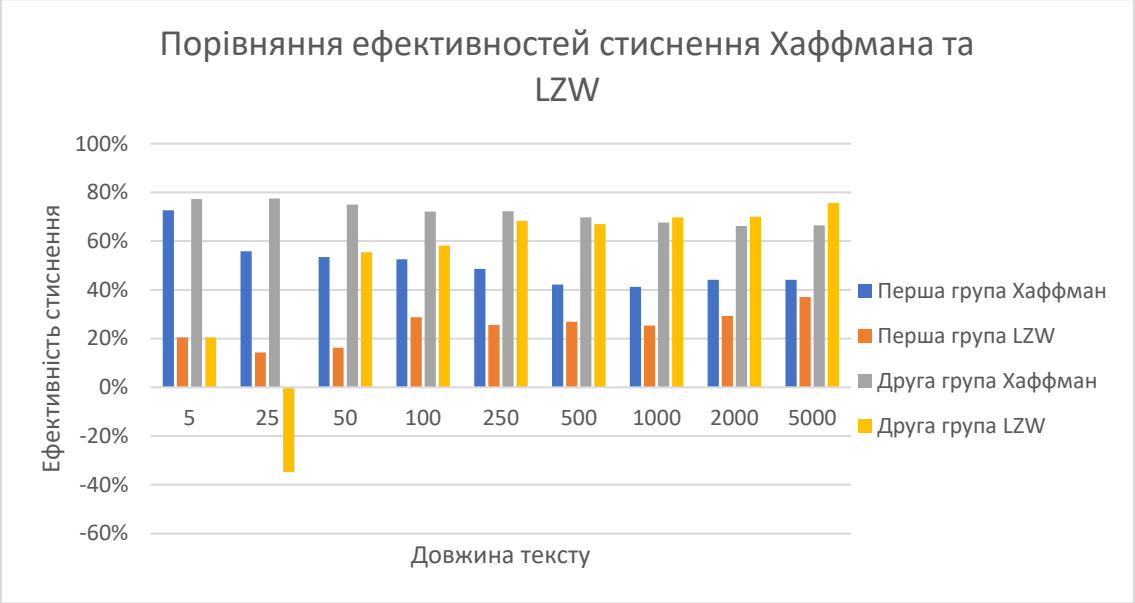


Рисунок 2.9 – Порівняння результатів тестування алгоритмів Хаффмана та LZW

На рисунку 2.9 видно, що алгоритм Хаффмана є найкращим інструментом для стабільного стиснення тексту. Хоча у деяких місцях кодування Хаффманом програє LZW, а саме при підвищенні довжини тексту, стабільність роботи алгоритму Хаффмана є головним плюсом цього алгоритму у реалістичних обставинах його використання.

2.2 Висновки за розділом

У цьому розділі було проаналізовано ефективність роботи алгоритмів Хаффмана та LZW на різних наборах даних, дивлячись на основні параметри текстових даних, що оброблюються. Було виявлено стабільну роботу алгоритму Хаффмана при більшості реалістичних змін у параметрах вхідного тексту.

3 РЕАЛІЗАЦІЯ АЛГОРИТМУ ХАФФМАНА У ПРОГРАМНОМУ ДОДАТКУ

3.1 Обирання інструментів для розробки та проєктування програмного продукту

Будь який процес розробки програмного продукту несе у собі необхідність проектування, створення прототипів та моделей, відносно яких буде розроблено фінальну версію продукту.

Для обирання інструментів для розробки та проектування програмного додатку – необхідно визначитися з основними задачами та властивостями програмного продукту.

Програмний додаток повинен мати інтуїтивний інтерфейс, який дозволить користувачеві закодувати вхідні текстові дані за алгоритмом Хаффмана, використовуючи вже існуючий ключ, або створити новий ключ і закодувати текст відносно нього. Елементи інтерфейсу повинні бути реалізовані таким чином, щоб користувач зміг запам’ятати усі необхідні кроки для звичайного алгоритму роботи з додатком за декілька перших ітерацій використання додатку. Уся зайва, для користувача, інформація не повинна бути відображена у стандартному стані програми. Користувач повинен мати змогу оцінити ефективність роботи програми за параметрами вхідних та вихідних даних за допомогою програмного продукту.

Відносно зазначених, основних, критеріїв, було визначено необхідність у таких типах інструментів для проектування та реалізації програмного продукту:

- інструменти для створення діаграм та схем;
- інструменти для розробки програмного продукту;
- система контроля версій.

Зазначені типи є основними інструментами для проєкту цього розміру. Інструмент для створення діаграм та схем може бути використаним для

створення блок-схем та простих макетів інтерфейсу програмного додатку. Існують спеціалізовані інструменти для створення та проєктування інтерфейсів програмних додатків, наприклад «Figma», але вони розраховані для проєктів, що мають необхідність або високий пріоритет у реалізації високоякісного та привабливого інтерфейсу, розробляємий продукт не має необхідності у цьому, через відсутність великої кількості елементів, які необхідно відображати у цьому додатку.

Тому, для цього проєкту було обрано програму «diagrams.net», яка більш відома як «draw.io» за старою назвою, яка дозволяє розроблювати велику, різноманітну, кількість схем та діаграм. Вона є безкоштовною та має відкритий доступ до свого коду. За допомогою цього додатку може буте створені необхідні макети інтерфейсу, блок-схеми алгоритмів та користувацьких сценаріїв, за необхідністю. Рисунок 1.2, що був наведений раніше, був розроблений за допомогою цього програмного додатку. «diagrams.net» може бути скачаний як окрема програма на персональний комп’ютер розробника, або може бути запущений на сайті, який відповідає назві продукту - «diagrams.net» або «draw.io». На рисунку 3.1 зображено інтерфейс цього інструменту з приклад розробленої схеми.

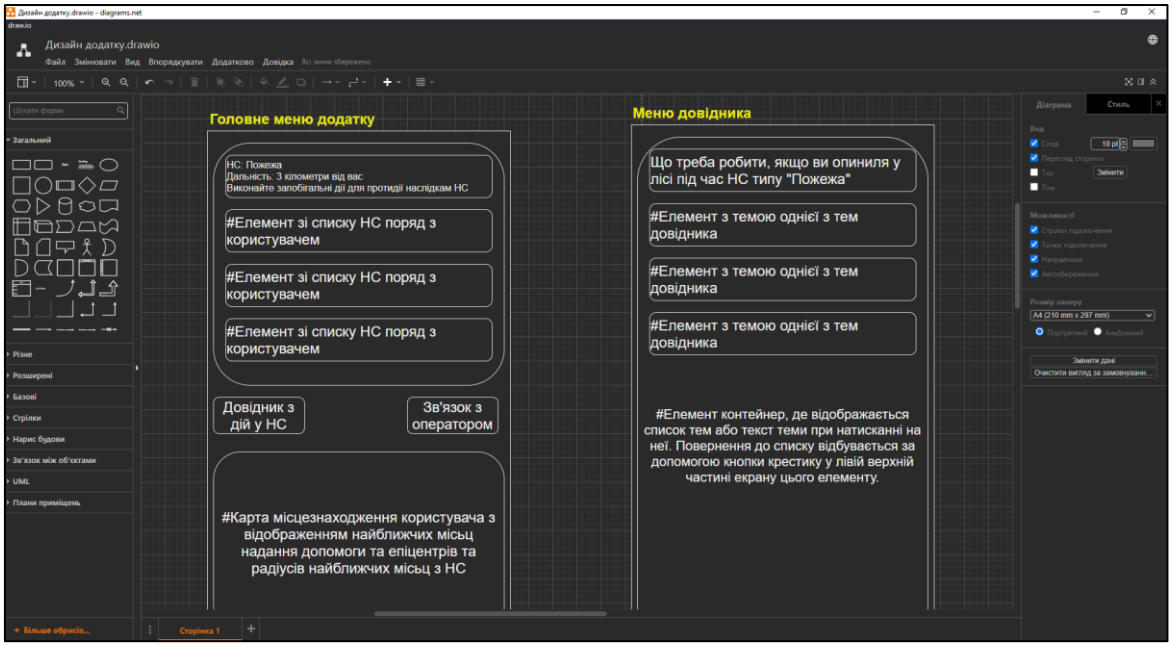


Рисунок 3.1 – Інтерфейс додатку «diagrams.net»

Інструментом для розробки програмного продукту обрано середу розробки «Microsoft Visual Studio». Так як мова програмування, яка буде використана для реалізації розроблюємого проєкту є «C#», це середовище буде найкращим для цього. «Microsoft Visual Studio» надає усі необхідні функції розробки програмних додатків з використанням мови «C#»:

- вбудований інструментарій для створення користувацьких графічних інтерфейсів;
- інструменти для компіляції програмного коду мови «C#»;
- можливість додавання сторонніх плагінів для використання додаткових функцій;
- підтримка найпоширеніших версій VCS.

За допомогою цієї середи розробки можливо реалізовувати програмні додатки на інших мовах програмування, наприклад «C», «C++», «Python». «Visual Basic» та інші, які пов’язані з компанією «Microsoft» або є поширеними та загальнодоступними мовами.

«Microsoft Visual Studio» також дозволяє створювати додатки розроблені на мові «C#» з реалізацією графічного інтерфейсу з допомогою бібліотек «Windows Forms» на базі фреймворку «.NET», через що програму можна встановити на усіх комп’ютерах де встановлена система «.NET Core», без необхідності у встановленні будь-яких додаткових бібліотек або систем, якщо вони не були застосовані у розробці проєкта. Приклад інтерфейсу додатку «Microsoft Visual Studio», на якому зображено процес розробки програмного коду та графічного інтерфейсу, зображено на рисунку 3.2.

Так як «Microsoft Visual Studio» є продуктом компанії «Microsoft», існує багато можливостей використання інших продуктів від компанії паралельно з цим за необхідністю підвищення ефективності розробки за допомогою інших інструментів, але для цього, частіш за все, необхідна додаткова плата. Сам «Microsoft Visual Studio» є безкоштовним за умовою некомерційного використання, тобто розробка програмних додатків з метою продажу програми або її продуктів потребує придбання ліцензії, якщо програмою

користується не фізична особа, а компанія або організаційна структура.

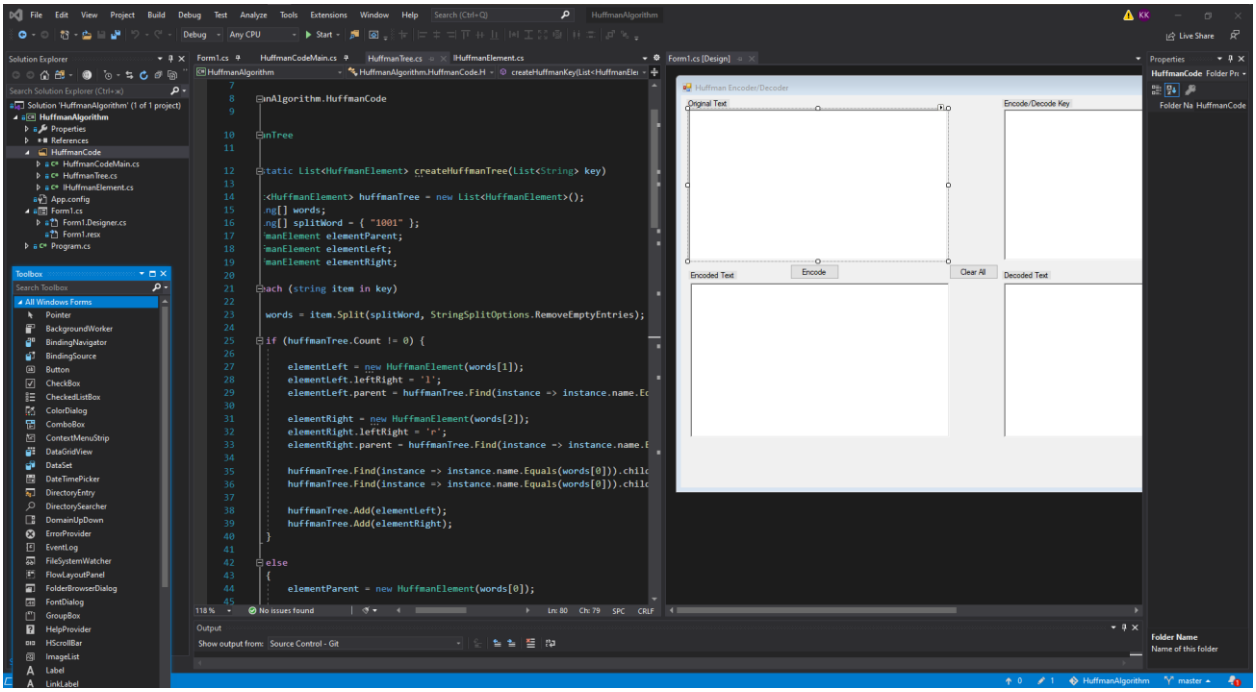


Рисунок 3.2 – Інтерфейс додатку «Microsoft Visual Studio»

Для системи контролю версій було використано систему «Git» зі створенням репозиторію у системі «GitHub», яка є найпопулярнішою та найпоширенішою системою для контроля версій програмних додатків. Більшість альтернатив є, здебільшого, копіями «GitHub», які створені тільки для подальшої інтеграції у паралельні продукти компанії-розробника, як, наприклад, система «BitBucket» яка є частиною конгломерату «Atlassian», яка володіє системами «Jira», яка є одною з найпопулярніших систем для менеджменту проєктів, та «Confluence».

«GitHub» є безкоштовним, для проєкту цього розміру, тому це є найкращим варіантом для використання при розробці.

Для використання системи «GitHub» буде використовуватися стандартний консольний додаток «Git», який дозволяє взаємодіяти з репозиторієм за допомогою консольних команд. Різниці між використанням вбудованого інтерфейсу взаємодії з VCS у «Microsoft Visual Studio» та консольною версію – немає, але для більш зрозумілого процесу взаємодії з

репозиторієм, використання базової форми є найкращим варіантом, через його «найнижчий» рівень реалізації, так як будь який інтерфейс для взаємодії з репозиторієм буде виконувати ті ж самі функції, але буде ховати усі внутрішні процеси за користувацьким інтерфейсом. На рисунку 3.3 зображено репозиторій, який знаходиться у системі «GitHub» для розроблюємого проєкту.

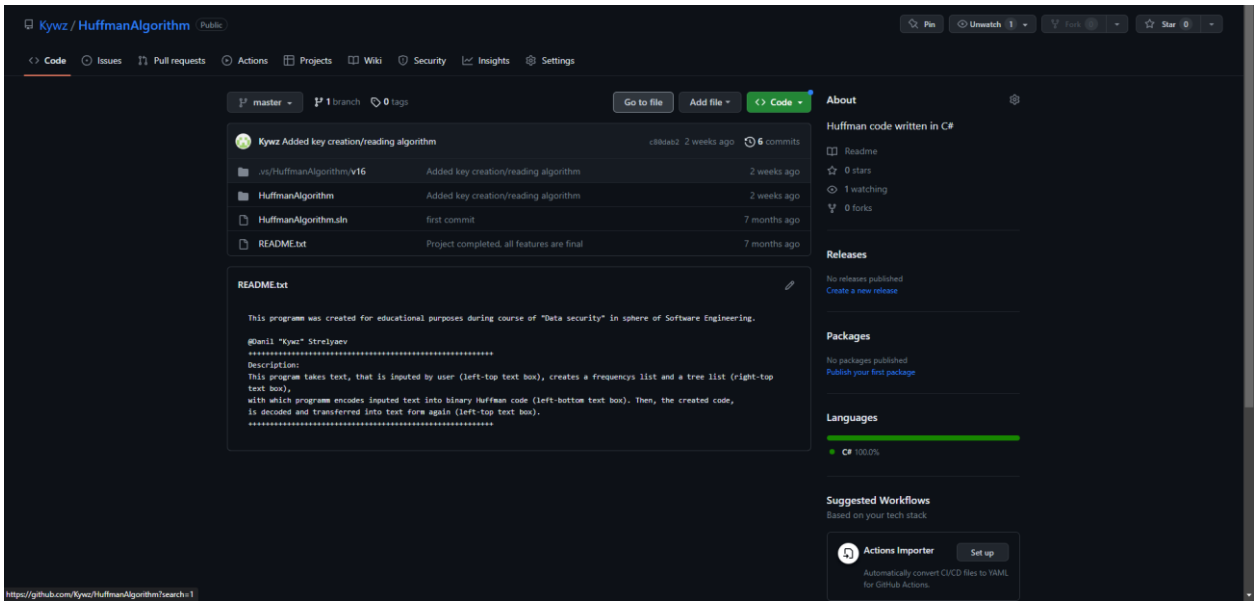


Рисунок 3.3 – Інтерфейс додатку «GitHub»

Усі зазначені інструменти будуть використані при розробці програмного забезпечення як для проєктування, так і для розробки, відповідно щодо їх можливостей. Усі елементи розроблюємого проєкту, окрім деяких елементів проєктування, будуть зберігатися у створеному репозиторії.

3.2 Реалізація програмного продукту за допомогою C#

Як було зазначено раніше – середою розробки буде програмний додаток «Microsoft Visual Studio». Програмний додаток та усі необхідні додаткові елементи було встановлено за вказівками, що були вказані у інсталятору, після чого був розпочатий процес розробки проєкту на мові «C#» з використанням бібліотек, пов’язаних з «Windows Forms».

Для створення проєкту, було запущено «Microsoft Visual Studio 2019»,

після чого було відображене стартове меню (рисунок 3.4), де було відображене функціональне меню з можливістю запуску створених проектів, за наявності, створення нового проєкту, шляхом імпорту проєкту або репозиторію та створенням нового проєкту.

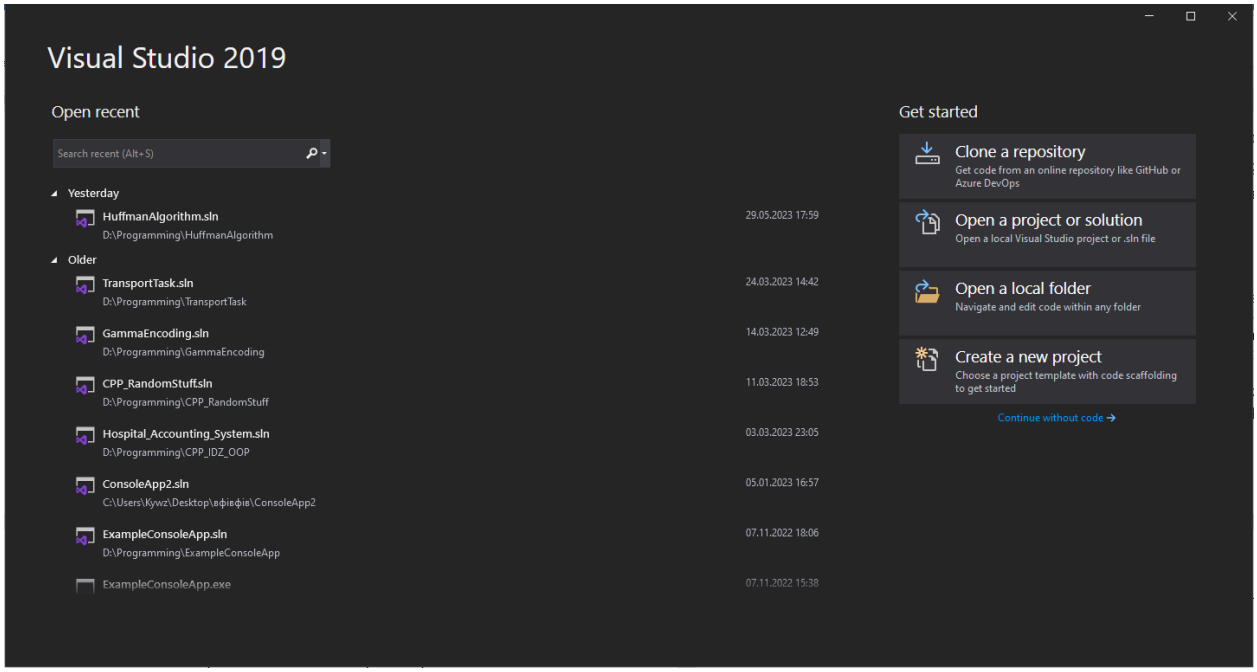


Рисунок 3.4 – Стартове меню додатку «Microsoft Visual Studio 2019»

Для початку роботи, було обрано варіант створення проєкту «Create a new project», після чого було обрано шаблон типу «Windows Forms App (.NET Framework)» на мові «C#», після цього були встановлені додаткові параметри, такі як назва та розташування проєкту у системі.

Після генерації шаблону проєкту, було отримано стандартний набір складових проєкту типу «Windows Forms App», у якому були файл конструктору форми, файл типу «Main» та функціональний файл для елементів форми.

Після створення проєкту, було також створено локальний репозиторій на комп’ютері розробника, у якому відбувалися усі взаємодії з системою «VCS» під час розробки.

Для створення початкового списку завдань для реалізації, було вирішено

створити прототип графічного інтерфейсу додатку, який буде доопрацьовуватися у майбутньому. Для цього необхідно визначитися зі переліком функціоналу додатку і створити усі необхідні елементи інтерфейсу наперед, або розробити інтерфейс таким чином, що додавання нових елементів не буде потребувати великої переробки існуючого інтерфейсу.

Відносно поставленого завдання було вирішено додати такі функціональні одиниці інтерфейсу:

- текстове вікно, де користувач буде вводити вхідні дані для шифрування за існуючим ключем або створюючи новий ключ;
- дві кнопки взаємодії з програмним кодом для шифрування або створення нового ключа для вхідного тексту у текстовому вікні вхідних даних;
- текстове вікно з даними, щодо процесу створення нового ключа для шифрування, з переліком символів, їх частотою та текстовою формою дерева Хаффмана;
- текстове вікно з вихідними даними у вигляді закодованого тексту;
- кнопка взаємодії з програмним кодом для дешифрування зашифрованого тексту у текстовому вікні з закодованими вхідними даними;
- текстове вікно з декодованим текстом;
- текстове вікно з дешифрованим текстом;
- текстове вікно з ключом для кодування;
- дві кнопки взаємодії з програмним кодом для зберігання та завантаження ключу шифрування у текстове вікно з ключом;
- текстові строки, які вказують на функції, що виконує елемент інтерфейсу;
- кнопка для відображення додаткової інформації про процес шифрування та дешифрування, у якому буде відображатися початковий розмір вхідних даних та розмір вихідних даних.

Після урахування усіх елементів інтерфейсу, які будуть реалізовані, було розроблено початкову форму інтерфейсу, яка зображена на рисунку 3.5, відносно якої буде розроблюватися функціонал програмного додатку. Також

було додано кнопку для очистки усіх вікон від тексту, яка полегшує процес роботи з програмою.

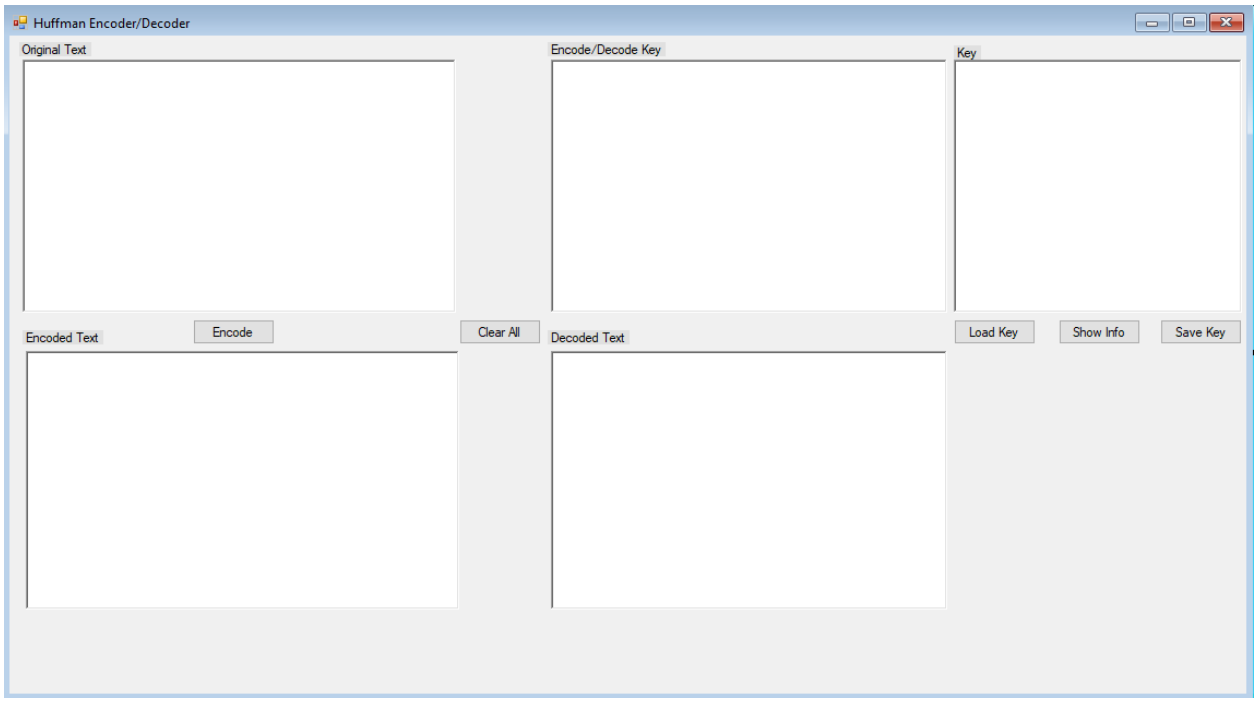


Рисунок 3.5 – Прототип графічного інтерфейсу розробленої програми

Текстові вікна, які були використанні у цьому прототипі, були побудовані за допомогою вбудованого об’єкту форм «RichTextBox», які дозволяють використовувати форматований текст та абзаци.

Після розробки макету додатку, було розпочато роботу у створені алгоритму взаємодії елементів інтерфейсу між собою, використовуючи замість реальних перетворень елементів для роботи алгоритму Хаффмана – заглушки.

Процес розробки графічного інтерфейсу та його функціоналу, а саме перевірка вхідних даних користувача, є таким же важливим, як і розробка самих алгоритмів обробки вхідних даних для створення вихідних, через те, що найбільшою причиною помилок у роботі програмного забезпечення завжди буде користувач, тому створення умов повного захисту програмного забезпечення від усіх можливих взаємодій користувача з додатком є одним з пріоритетних завдань.

Спочатку було розроблено алгоритм переведення тексту з вікна для вхідних даних у змінну для подальшої обробки. Цей алгоритм потребує реалізації перевірок на обмеження, що не дають ввести користувачеві текст, який має у собі тільки один символ, не дивлячись на кількість повторень цього символу. Це необхідно, через марність перетворення при таких умовах, тому що закодований текст не буде мати сенсу з точки зору кодування Хаффманом, бо результат буде відповідати перетворенню символу у тексті на нуль або один, у деяких випадках ефективнішим буде алгоритм кодування RLE, який перетворить текст з, наприклад, повторюючимися символами «a» кількості разів «n» у текст форми «na», тому у цьому випадку ця перевірка та обмеження є доцільними.

Так як завданням цього додатку є шифрування тексту, інших обмежень для програми реалізовувати нема необхідності.

Після створення алгоритму перетворення користувацького введення тексту у вхідні дані для програми, було розпочато розробку алгоритму організації даних. Для цього було використано технології інтерфейсів.

Інтерфейси – абстрактні класи, що дозволяють зберігати у собі абстрактні методи та властивості. Практичне їх використання у цьому проєкті – створення користувацької «змінної», що буде виконувати роль вузлу дерева, зберігати методи для обробки даних для цього інтерфейсу та зберігатиме певну інформацію, що дозволить виконувати алгоритм Хаффмана.

Було розроблено інтерфейс «IHuffmanElement», який зберігає 5 змінних:

- «string name», назва вузлу;
- «int frequency», частота вузлу;
- «char leftRight», булеве значення, що відповідає місцезнаходженню вузлу відносно інших;
- «HuffmanElement parent», посилання на батьківській вузол;
- «List<HuffmanElement> child», посилання на вузли, які походять від оброблюємого вузла.

Необхідно зазначити, що для змінної «leftRight» краще, дивлячись на оптимізацію, визначити як тип даних «bool», але самий вищий вузол, який має найбільшу частоту, не може бути зліва або справа, тому тип «bool» не може бути використанням для цієї змінної.

Після створення інтерфейсу, було імплементовано клас, який походить від інтерфейсу (рисунок 3.6), для якого було створено декілька методів, які дозволять створювати нові ітерації цього класу від різних типів вхідних даних та різні допоміжні методи обробки та виведення даних.

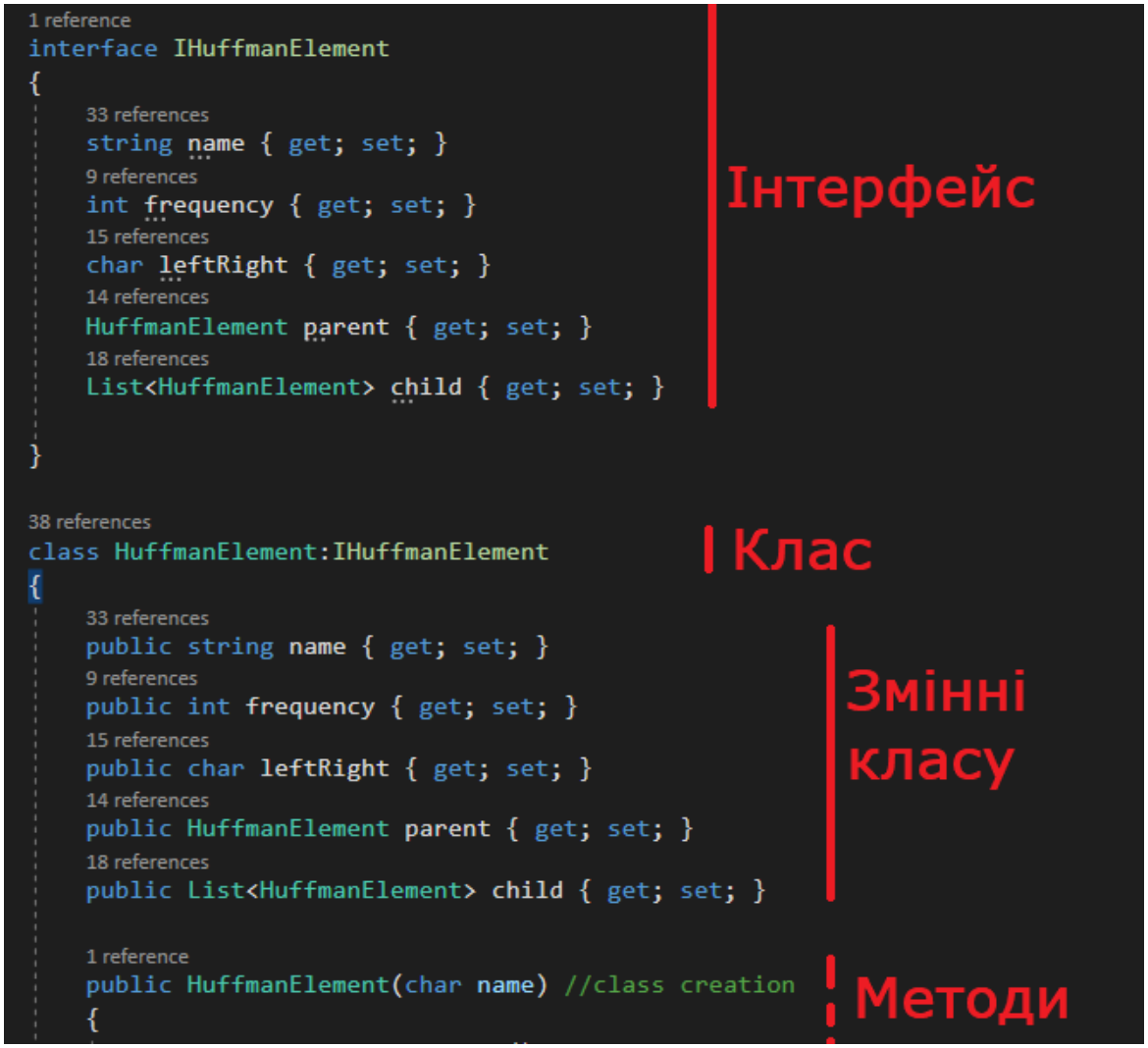


Рисунок 3.6 – Створення класу через імплементацію інтерфейсу

Треба зауважити, що параметри «parent» та «child» виконують роль з’єднуючих граней, які поєднують елементи між собою посиланнями на одне-

одного.

Після створення інтерфейсу і реалізації усіх допоміжних методів, було розпочато роботу над перетворенням вхідного тексту до елементів дерева Хаффмана. Для цього необхідно створити алгоритм перетворення вхідного тексту у набір ідентичних символів та їх частоту. Цей алгоритм було реалізовано як простий ітератор, що бере перший символ з вхідного рядка, починає інший ітератор, який рахує загальну кількість символів у рядку, після чого видаляє усі повторення оброблюємого символу у рядку, закінчує роботу другого ітератору та починає наступний крок першого ітератору. Для реалізації цього алгоритму, використано метод мови «C#» - «String.Replace». Блок-схема, що описує принцип роботи цього алгоритму зображена на рисунку 3.7.

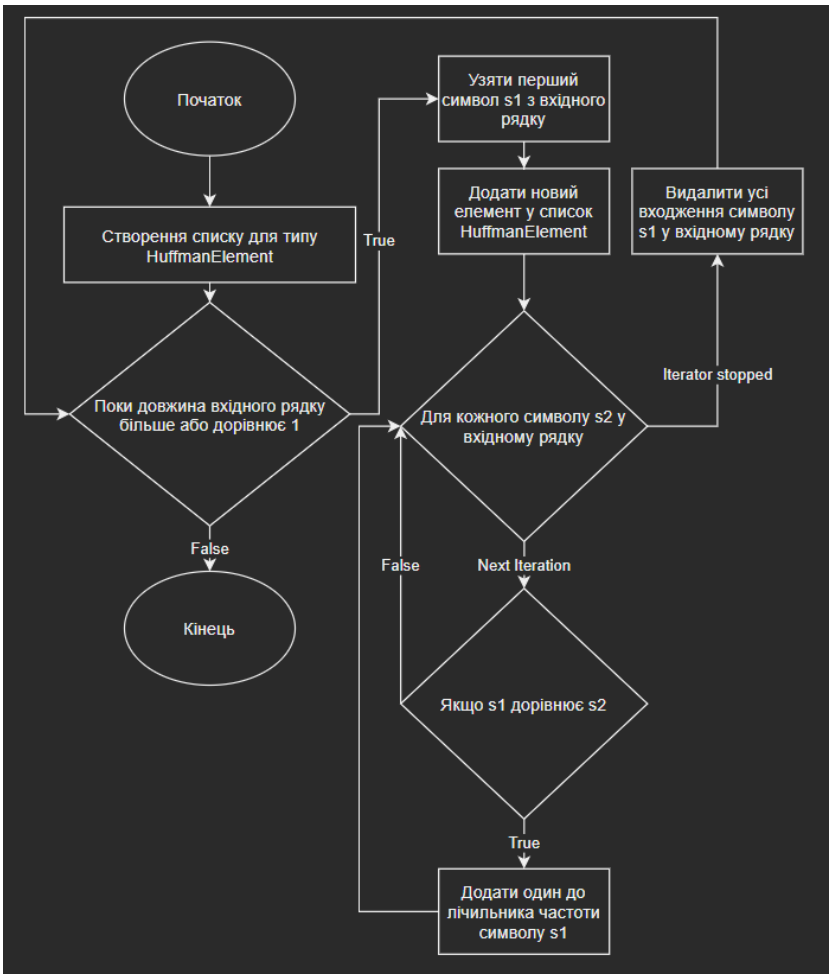


Рисунок 3.7 – Блок-схема роботи алгоритму створення варіанту таблиці частот для алгоритму Хаффмана з використанням методів «C#»

Також існує можливість реалізації цього алгоритму через видалення усіх повторювань першого символу з рядку зі збереженням попередньої версії, що дозволить відняти розмір відредагованого рядка від необробленого, таким чином знаходячи частоту видаленого символу. Це рішення може виглядати більш оптимізованим, за обране, але лише зі сторони кількості використовуємих розрахункових можливостей машини, що виконує цей алгоритм.

Пам’ять, у випадку використання алгоритму пошуку різниці, у критичний момент навантаження, на першому кроці алгоритму, буде займати подвійний розмір пам’яті вхідного тексту, тому при обробці даних великого обсягу – можуть з’являтися певні проблеми. Також, алгоритм видалення елементів з рядку може повністю відповідати реалізованому за параметрами оптимізації, перевірити це не є можливим через закритого доступу до коду реалізації методу. У такому випадку – реалізований програмний код буде максимально оптимальним. Тому, у цьому випадку, було вирішене оптимізувати розмір використовуємої пам’яті, замість розрахункових ресурсів.

Після реалізації алгоритму у вигляді програмного коду та організації взаємодії графічного інтерфейсу додатку з програмним кодом, було перевірено роботу алгоритму на простому прикладі, де ручна перевірка не займе багато часу. Результат роботи програмного додатку зображено на рисунку 3.8. Повна таблиця частот зображена на рисунку 3.9.

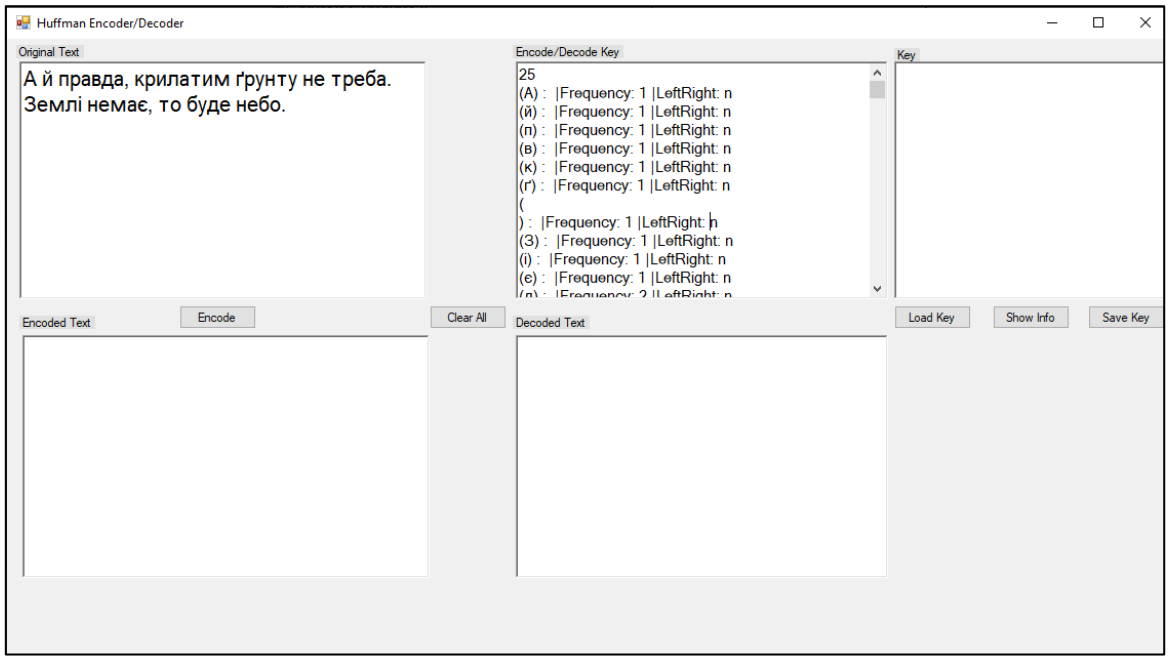


Рисунок 3.8 – Результат роботи програмного додатку у створення таблиці частот

Загальний формат побудови таблиці частот має такі параметри: «(СимволОбробки): |Frequency: Частота |LeftRight: ЗначенняЛівоПраво». Ці параметри генеруються під час створення таблиці частот. Самий перший рядок у вікні «Encode/Decode Key» є лічильником усіх ідентинтичних символів у вхідному тексті.

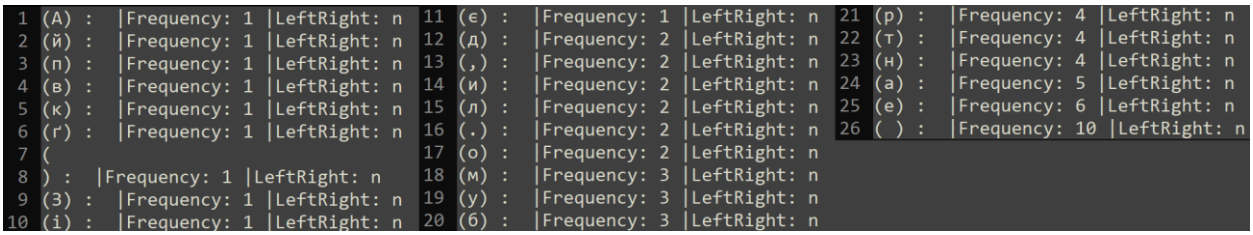


Рисунок 3.9 – Згенерована таблиця частот

Ручна перевірка показала правильність згенерованих результатів.

Після розробки алгоритму створення таблиці частот, було почато розробку алгоритму створення дерева Хаффмана, для подальшого кодування вхідного тексту у зашифрований вигляд.

Для цього, отриману таблицю частот, у вигляді списку, було

відсортовано у порядку зростання за допомогою методу «List.OrderBy()». Після цього, було розроблено алгоритм, відносно вхідних даних. Блок-схема алгоритму зображена на рисунку 3.10.

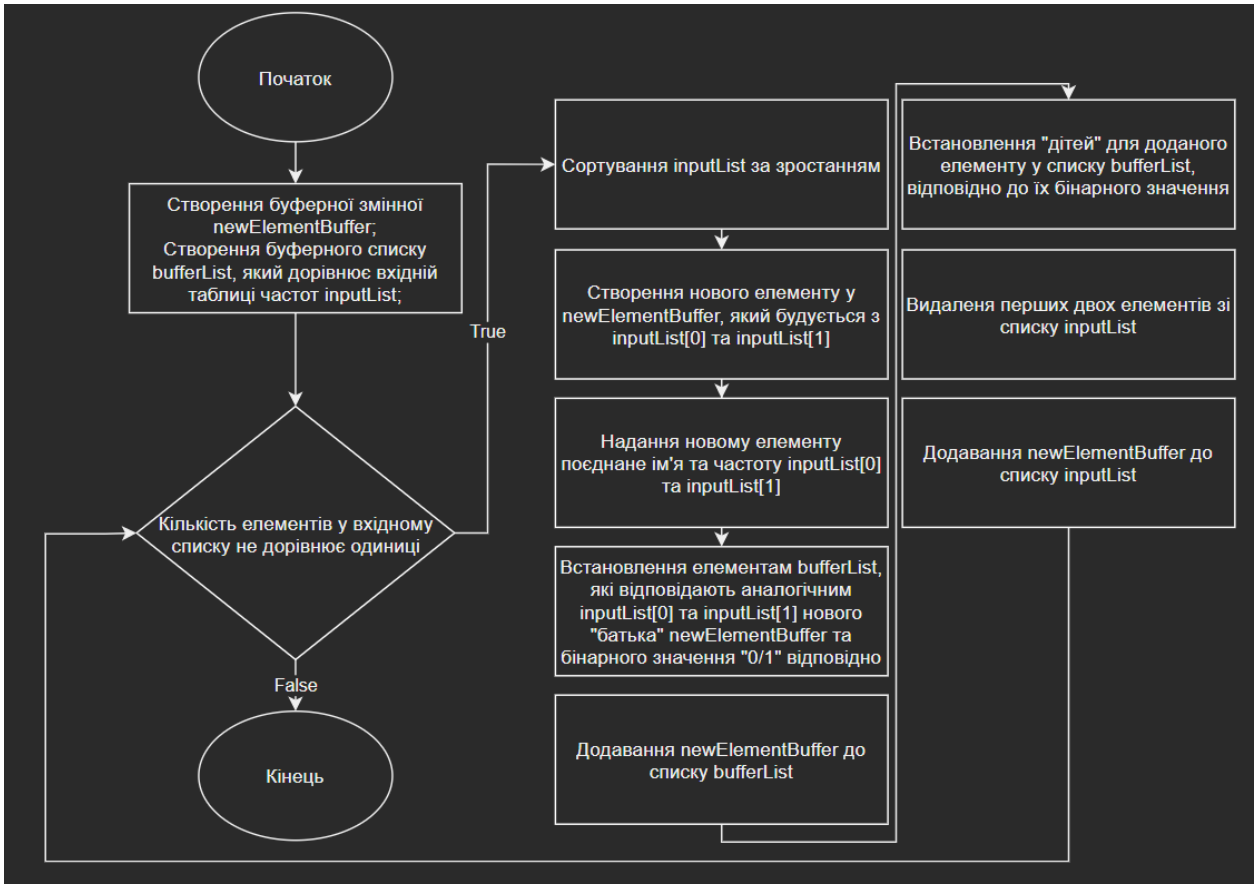


Рисунок 3.10 – Блок-схема роботи алгоритму Хаффмана у розробленому додатку

Слід зауважити, що алгортим, описаний на рисунку 3.10 розроблений саме для відображення роботи алгоритму у розробленому додатку, а не загальний принцип роботи цього алгоритму. Методи реалізації представлення даних мовою «C#», наприклад, використання списків та їх методів для сортування, дозволяють створити більш простий у розумінні та реалізації алгоритм. При сортуванні списку, алгоритм завжди буде мати необхідні йому значення наприкінці або на початку списку, залежно від методу сортування, що виключає необхідність у ручному сортуванні, а використанні методів інтерфейсу дозволяє зберігати усі змінні, з різними типами даних, у одній

змінній, яку можна зберігати у списку, замість масиву або декількох різних масивах, які створені під кожний параметр вузлу дерева. Таким чином, технології та методи мов програмування дозволяють перероблювати принцип роботи та саме представлення різних алгоритмів і їх практичних реалізацій.

Реалізувавши цей алгоритм у вигляді програмного коду, до функціоналу програми було додано можливість створення дерева Хаффмана, за допомогою якого вже можна проводити шифрування. Для виведення інформації про вузли дерева Хаффмана, було розроблено метод для інтерфейсу, що виводить усі параметри вузла. Результат роботи програми з використанням нового функціоналу зображено на рисунку 3.11. Так як ці записи є доволі великими, було обрано текст маленького розміру для більш компактного результату.

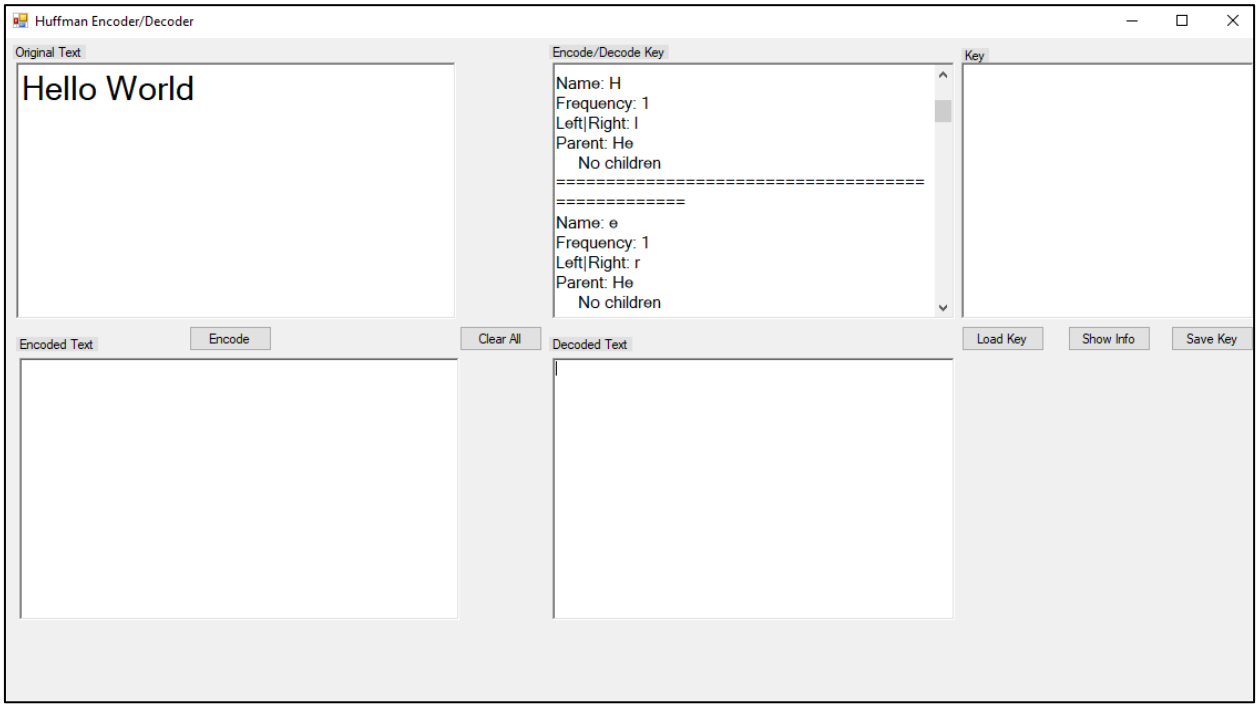


Рисунок 3.11 – Результат роботи додатку, при створені дерева Хаффмана

Повний список згенерованого дерева зображено на рисунку 3.12. Перевіривши згенерований результат з декількома реалізацій цього алгоритму у Інтернеті – було отримано такий самий результат, за ефективністю. Через різне сортування та реалізації цього алгоритму – дерево Хаффмана не завжди будується однаково на однакових текстах, тому, хоч вузли дерева мали різні

імена та, рідше, різні частоти, але усі вони надавали можливості зашифрувати та дешифрувати текст за допомогою створеного дерева.

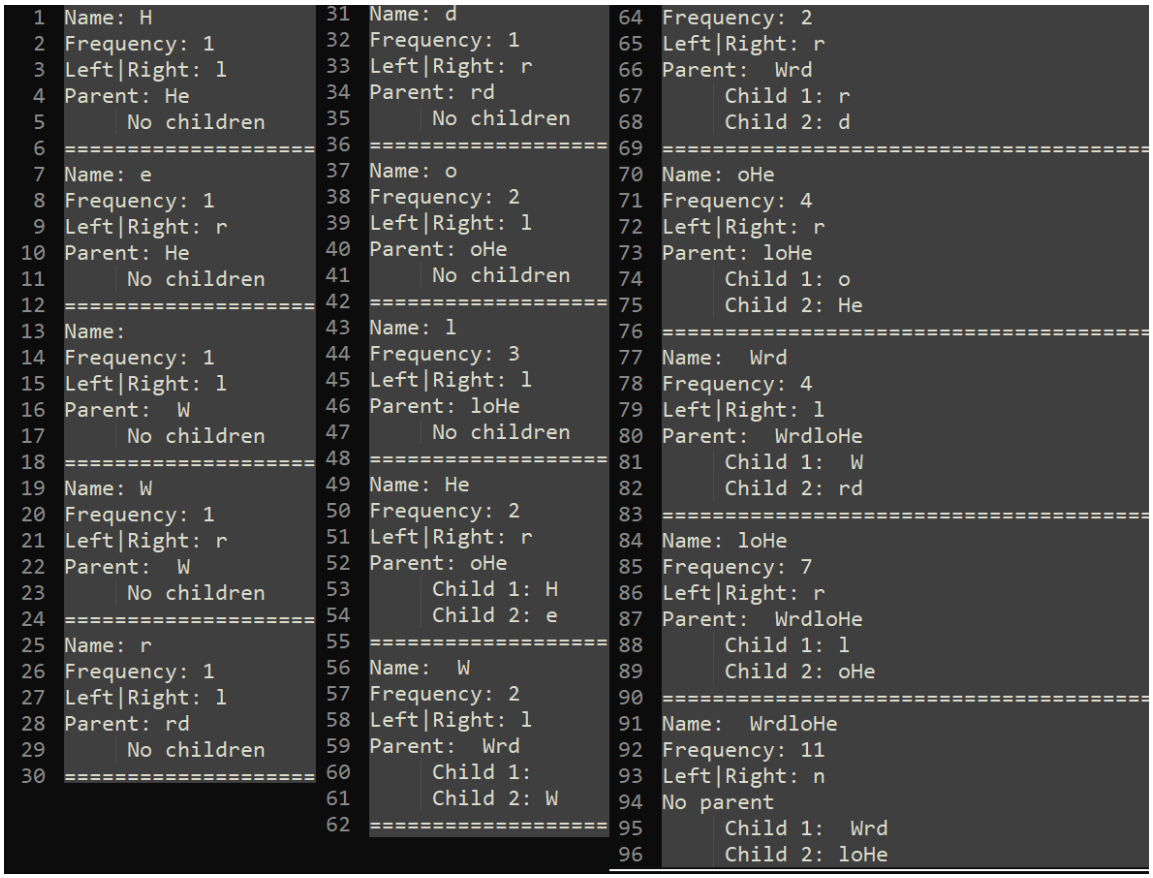


Рисунок 3.12 – Створене дерево Хаффмана для рядку «Hello World»

Створивши бінарне дерево Хаффмана, можливо виконати переведення вхідного тексту у зашифровану, бінарну, форму. Для цього необхідно розробити алгоритм, який буде читати кожний символ вхідного рядка, та перетворювати їх на бінарну форму за деревом Хаффмана. Для цього, при читанні символу, у згенерованому дереві буде відшукуватися його еквівалент, де, відносно параметру інтерфейсу «leftRight» цього елементу та його «батьків» буде будуватися бінарна форма запису символу.

Для цього було створено рекурсивний метод інтерфейсу, який зчитує наявність «батька» у елементу та повертає бінарне значення у рядковій формі, відносно запису елементу, але у випадку наявності батька у елемента, цей самий метод виконується для його батьківського елементу, та його результат

додається до попереднього. Цей алгоритм дозволяє знаходити кодування для кожного окремого вузлу дерева, але цю інформацію також треба перевести у бітову форму даних. Для цього був створений метод, який оброблює кожний символ, переводить його у бінарний вид у рядковому типу даних, після чого, результат роботи переведення зчитується та переводиться у бінарний тип даних. У кінці, цей результат виводиться на екран користувачеві.

Результат роботи програми у створені бінарного шифрування для вхідного тексту зображено на рисунку 3.13. Також, до інтерфейсу екрану було додано два рядки, які відображують бітовий розмір вхідного тексту та розмір закодованого вхідного тексту.

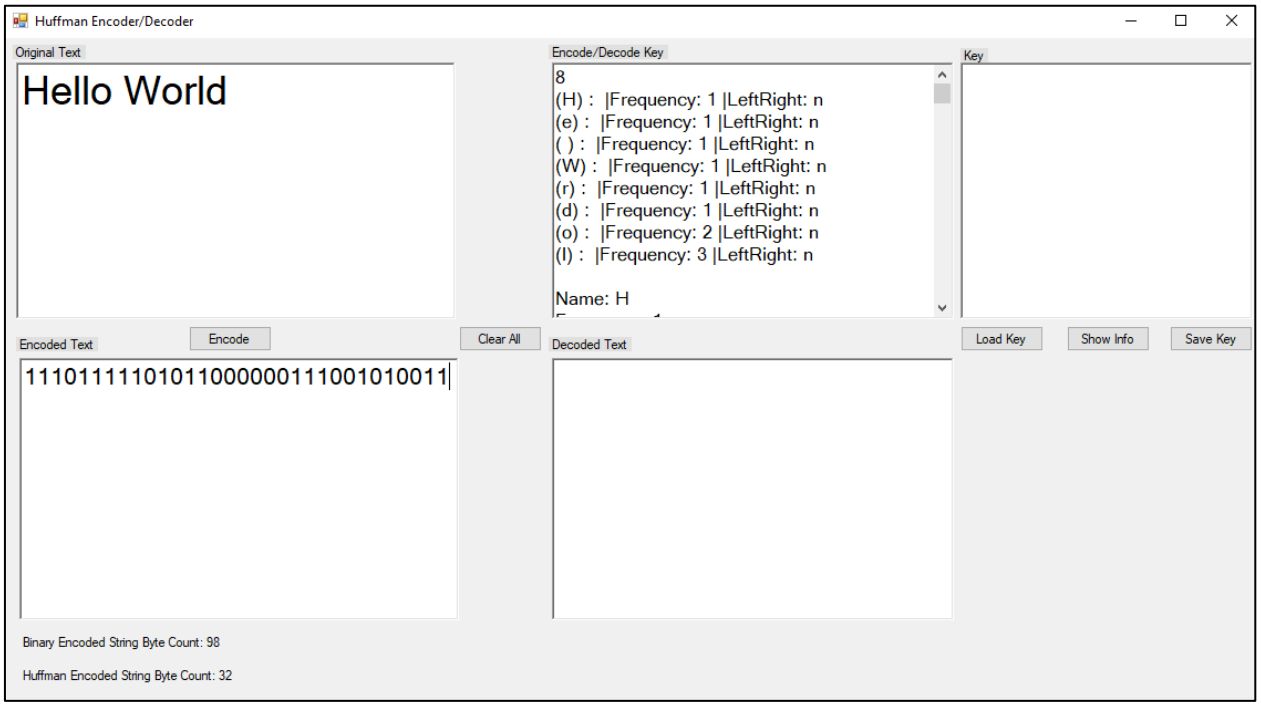


Рисунок 3.13 – Результат роботи програмного додатку при перетворенні вхідного тексту у бінарний вигляд

Розрахунок розмірності виконувався завдяки методам «C#», але слід зауважити, що при підрахуванні кількості байтів у вхідному рядку, використовується таблиця кодування «ASCII», тому результат підрахування розміру тексту може варіюватися при зміні кодування тексту. Але навіть враховуючи це – розмір закодованого тексту займає на 67% менше місця, аніж

оригінальний текст.

При ручній перевірці – результат роботи програми є вірним, але це не гарантує стовідсоткову правильність роботи програми, через малий об’єм вхідного тексту, але перевірити роботу цього алгоритму на більш великих об’ємах не є можливим, бо час на його перевірку буде надто великим, тому необхідно переходити до наступного етапу розробки – створення алгоритму дешифрування, тільки при його реалізації можна буде точно побачити правильність роботи розробленого алгоритму.

Для реалізації алгоритму дешифрування, необхідно реалізувати системи зчитування ключа, але під час розробки, можна використовувати вже згенероване дерево для дешифрування, одразу після його початкової генерації. Таким чином можна визначити коректність роботи алгоритму шифрування та дешифрування без реалізації додаткових елементів, які не обов’язково необхідні на цьому етапі.

Було реалізовано прости алгоритм сходження, де з самого вищого елементу дерева прокладається шлях до елементу, якій відповідає зашифрованому тексту, так само як і в ручній реалізації алгоритму, що була описана у першому розділі.

Після під’єднання усіх новостворених функціональних елементів коду (рисунок 3.14) до графічного інтерфейсу, було перевірено роботу програмного додатку на прикладу з першого розділу, так як він є більшим за рядок «Hello World», та вже був прорахований у ручному застосуванні цього алгоритму. Результат роботи програмного додатку при дешифруванні тексту зображено на рисунку 3.15.

```
public static string Decode(BitArray bits, List<HuffmanElement> HuffmanTree)
{
    HuffmanTree = HuffmanTree.OrderBy(instance => instance.name.Length).ToList(); // Sorting tree by name length, for no further use of frequencies
    string output = "";
    HuffmanElement buffElement = HuffmanTree.Last();
    foreach (bool b in bits) // Iterator for looking through encoded text
    {
        if (b == false)
        {
            buffElement = buffElement.child.ElementAt(0); // Getting left child from parent
        }
        else if (b == true)
        {
            buffElement = buffElement.child.ElementAt(1); // Geting right child from parent
        }
        if (buffElement.name.Length == 1) // Got to end of tree, length of name == 1 says that there is no next element possible
        {
            output += buffElement.name;
            buffElement = HuffmanTree.Last();
            continue;
        }
    }
    return output;
}
```

Рисунок 3.14 – Програмний код для дешифрування зашифрованого тексту

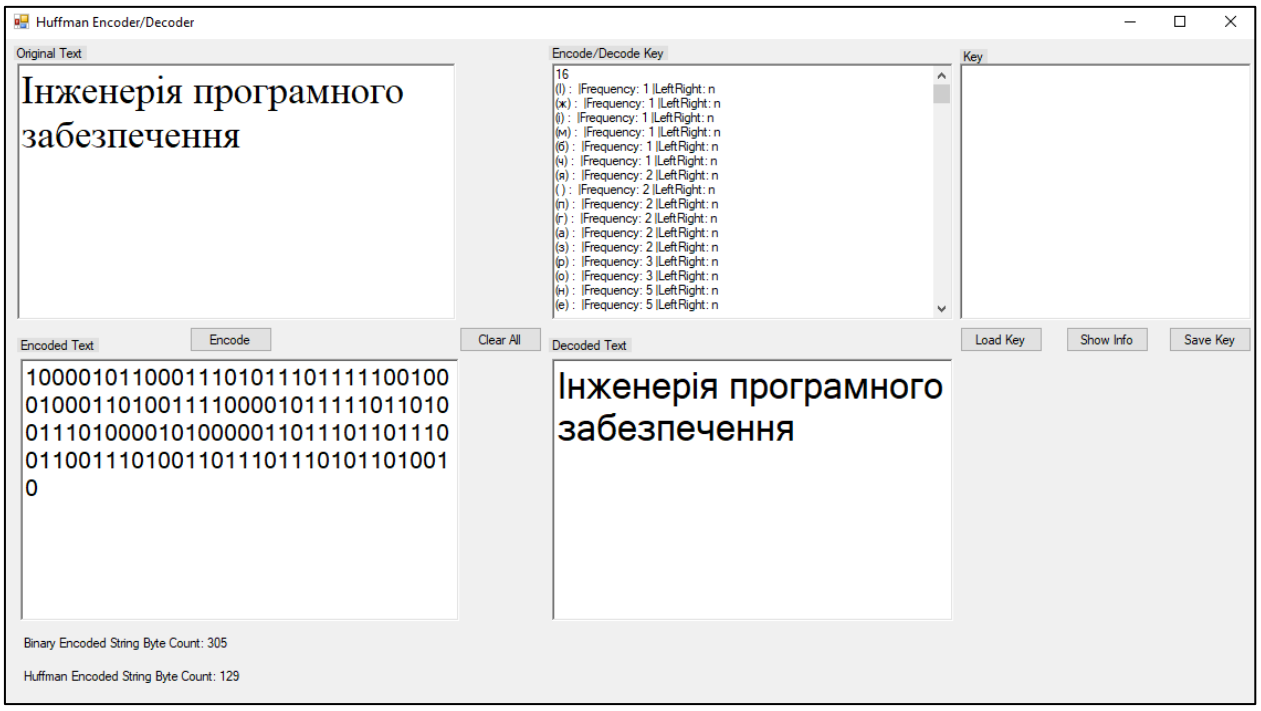


Рисунок 3.15 – Результат роботи програмного додатку при дешифруванні зашифрованого тексту

Як можна побачити, результат дешифрування повністю відповідає вхідному тексту. Рисунок 3.14 відображає принцип роботи дешифрування, у якому нема необхідності у будь-якій інформації, окрім назви елементу дерева та його посилання на його дітей, створюючи, таким чином, текстове представлення дерева Хаффмана, принцип якого буде використано для створення алгоритму генерації ключа для дешифрування, яке відрізняється від

стандартних методів реалізації цього елементу системи.

Так як вхідний текст, що шифрується, ніяк не пов’язаний з вихідним, що був дешифрований, можна точно сказати, що створений алгоритм побудови таблиці частот, бінарного дерева, шифрування та дешифрування працюють правильно, як система. Через це можна почати тестування програмного додатку на більших об’ємах даних, для перевірки стабільності роботи алгоритму при збільшенні об’єму даних. Під час розробки та тестування програмного додатку на етапі реалізації алгоритму дешифрування, було помічено залежність об’єму на правильність вихідних даних при наявності дефектів у коді, як, наприклад рядок з п’яти різних символів був зашифрований та дешифрований без проблем, а рядок з 15 різних символів не був нормально дешифрований, через що результат був спотворений, тому є необхідність у цьому кроці для виключення можливості спотворення даних розробленим програмним додатком. На рисунку 3.16 зображено перевірку роботи додатку при шифруванні тексту поеми Івана Котляревського «Енеїда». Для порівняння вхідного та вихідного тексту, було додано перевірку на відповідність текстів, яка, при виконанні умови відповідності, видає повідомлення про сходження текстів у новому текстовому вікні «Дебаг інформація» у нижньому правому куті графічного інтерфейсу, метод що використовується є стандартним методом «C#». Довжина тексту складає 359719 символів.

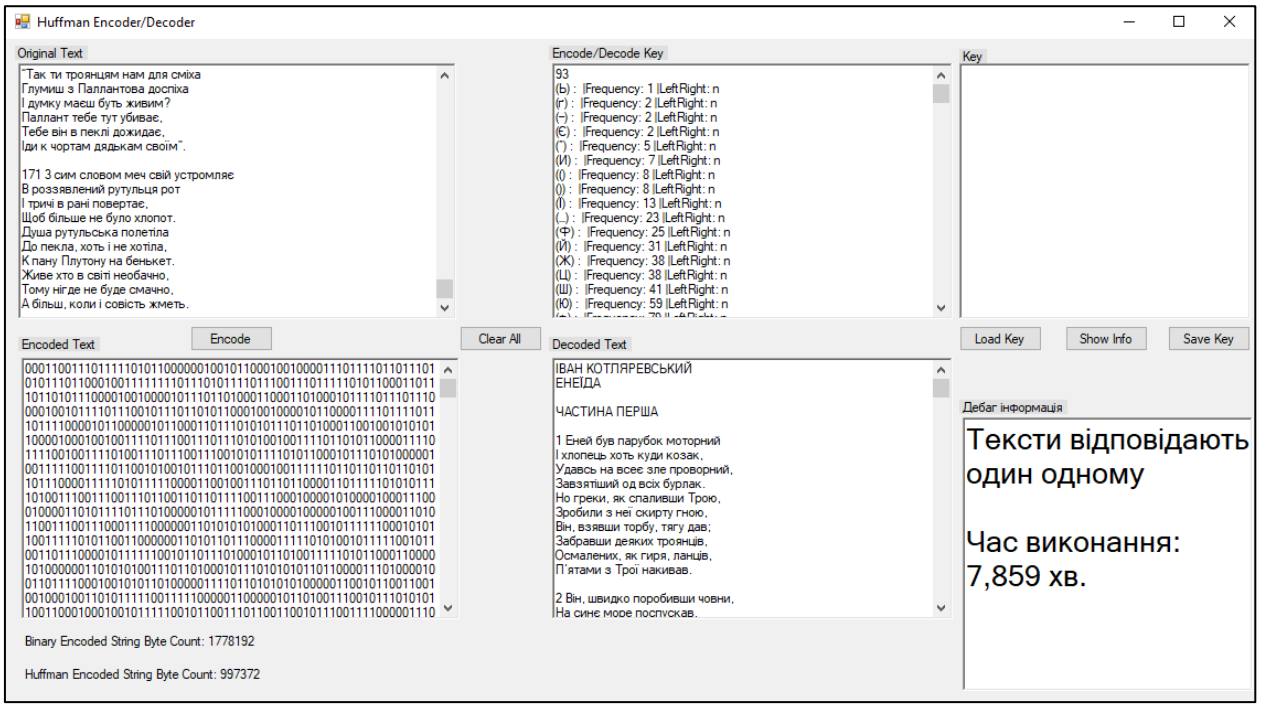


Рисунок 3.16 – Результат роботи додатку при шифруванні великого обсягу даних на першому ПК

Після кінця роботи алгоритму, було виведено повідомлення про відповідність текстів та час виконання алгоритму шифрування. Це тестування проходило на персональному комп'ютері, характеристики якого зазначені на рисунку 3.17.

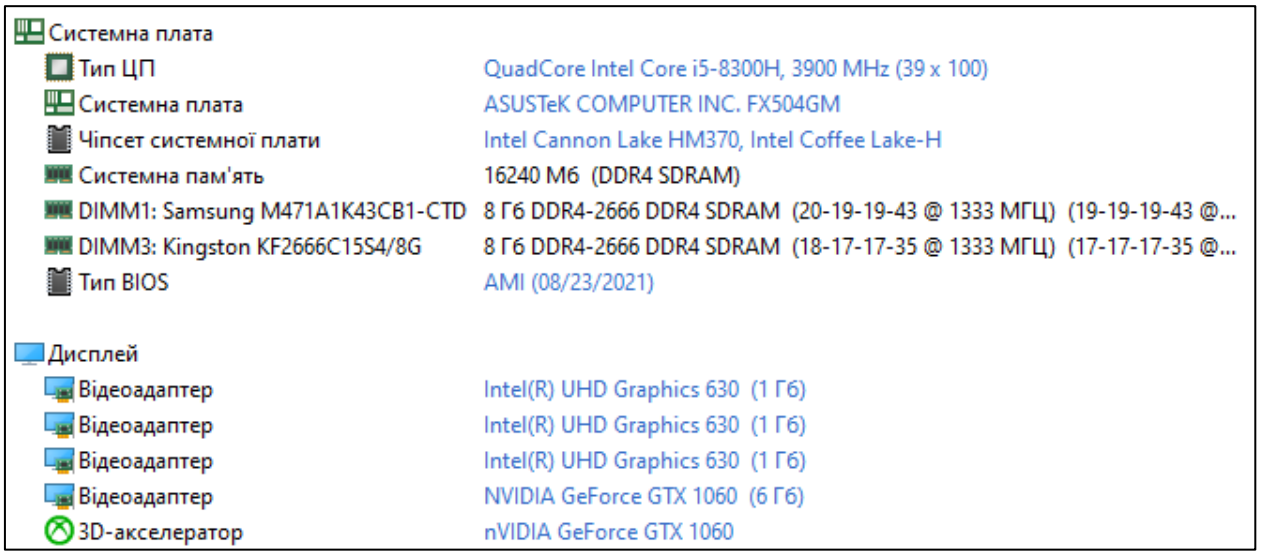


Рисунок 3.17 – Характеристики першого ПК, який виконував розроблений алгоритм шифрування

Для перевірки зміни часу виконання алгоритму на більш продуктивному ПК з більш просунутим фізичним обладнанням та новішою оперативною системою Windows 11, було виконано алгоритм шифрування з текстом з першої перевірки роботи додатку. Результат роботи додатку зображено на рисунку 3.18.

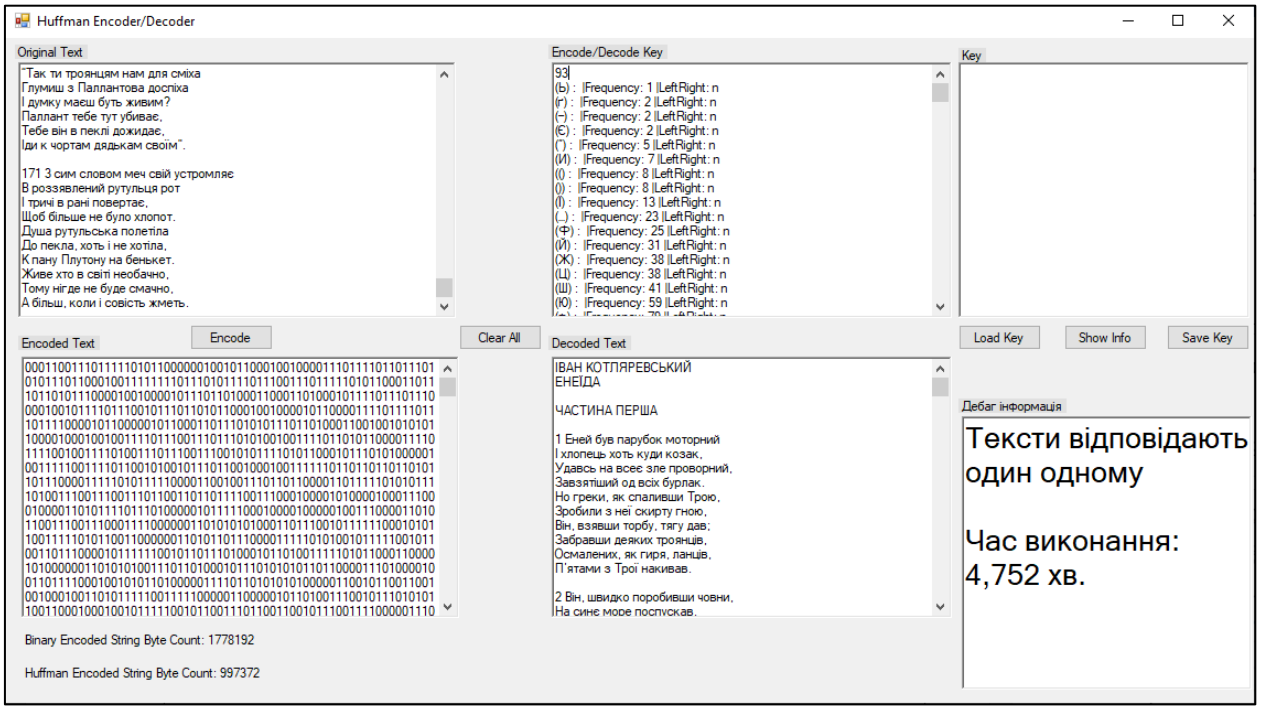


Рисунок 3.18 – Результат роботи додатку при шифруванні великого обсягу даних на другому ПК

На рисунку 3.19 зображено характеристики другого ПК.

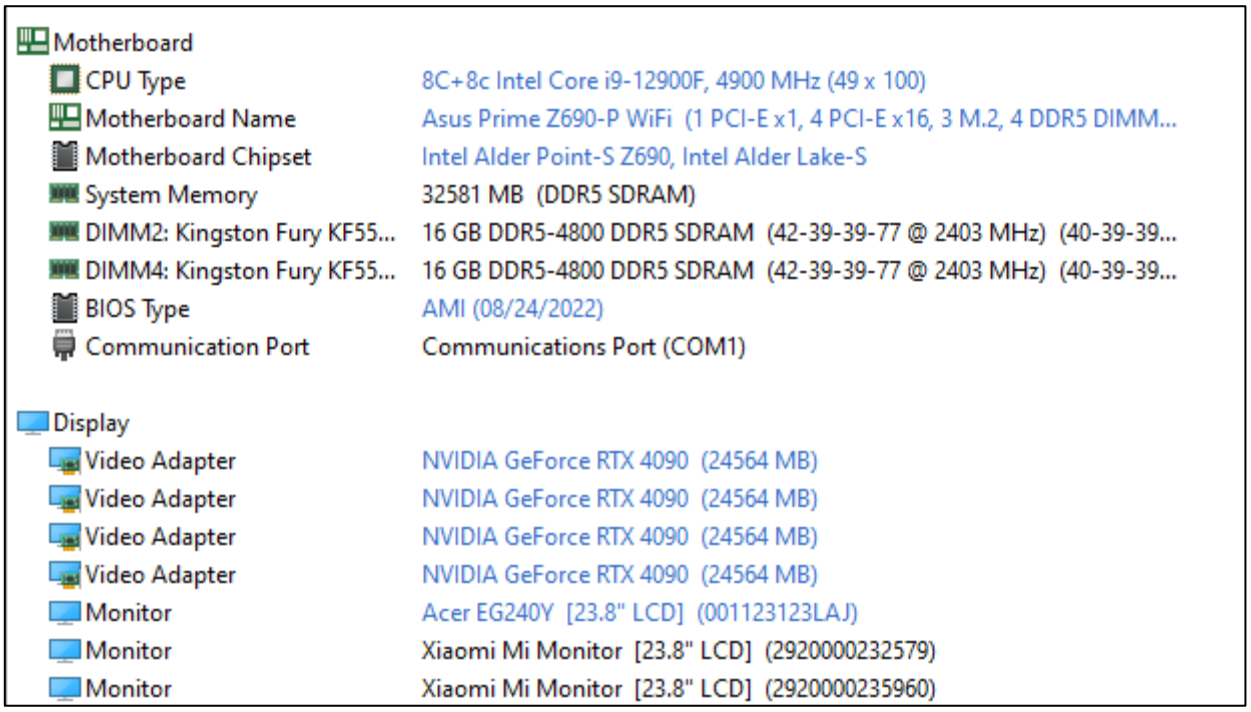


Рисунок 3.19 – Характеристики другого ПК, який виконував розроблений алгоритм шифрування

Два тести показали, що збільшення розрахункової потужності зменшує час виконання алгоритму при однакових умовах вхідний даних та показала однаковий результат вихідного тексту на двох різних системах. Таким чином, стабільність роботи основного алгоритму додатку є достатньою, та не потребує подальших змін.

Після реалізації алгоритму дешифрування, необхідно реалізувати алгоритм генерації ключа дешифрування та його зчитування.

Для цього, необхідно визначитися з синтаксисом генеруємого ключа. Зазвичай, для збереження інформації про зв’язки використовуються таблиці суміжності, для зберігання дерева без необхідності додаткового перетворення, та таблиці частот, але її використання потребує використання алгоритму створення дерева. Обидва цих алгоритмів потребує використання таблиць, для зчитування яких необхідні більш тяжкі, для реалізації та використання, алгоритми. Використання генеруємих вузлів дерев, що відображаються під час шифрування тексту також не є ефективним, через велику кількість повторюваних ключових слів, по типу «Ім’я». Також, повстає проблема

зчитування створеного ключа без спотворення вихідного тексту, через відсутність обмежень на використання будь-яких символів, що не дає можливості використання стандартних роздільників, таких як пробіл, новий рядок або один нестандартний для тексту символ.

Враховуючи усі проблеми, які можуть повстати при використанні інших методів створення ключів для шифрування, було розроблено систему, що генерує ключі у нестандартному вигляді, як рядок, який має послідовність з трьох елементів: «батько/дитина#0/дитина#1». Зберігання ключа у цьому вигляді для кожного елемента дерева Хаффмана дозволяє використання простих алгоритмів зчитування для цього ключа, та більш «незрозумілу» версію ключа, яка може здатися простим набором символів без прямого вказування на метод шифрування, для якого він застосовується, це додатково підвищує безпеку алгоритму.

Синтаксис елементу ключа, у загальному не має ніяких проблем, але як було зазначено раніше – без використання обмежень на символи, що можуть бути зашифровані, використання стандартних символів розділення не є можливим, так як усі вони можуть бути використанні у тексті, якій шифрується, через що процес роботи алгоритму зчитування ключа може бути порушений у деякий випадках. Через це, було обрано роздільник «1001», який не може бути випадкового відтворений у будь якому-випадку, при шифрування кожного символу окремо.

Роздільник «1001» дозволяє шифрування будь-яких символів, через відсутність сценаріїв випадкового самовідтворення цього коду, усі сценарії можливих генерацій відображені нижче, де роздільник виділений дужками, для кращого відображення послідовностей символів:

- «…10(1001)10…»;
- «…10(1001)01…»;
- «…01(1001)10…»;
- «…01(1001)01…».

Цей варіант роздільника є найменшим із можливих, так як усі інші

варіанти можуть бути випадково відтворенні, через що кінцевий результат дешифрування буде спотворений. Так як у тексті можуть бути символи, що є кодом переходу на наступний рядок, необхідно визначення символу закінчення рядку, їм може бути модифікований рядок «1001» – «101101», який теж не може бути випадково відтвореним.

Алгоритм створення ключа не виконує ніяких алгоритмів, відбувається поєднання назви батька та його дітей з використанням роздільника.

Після розробки програмного коду для генерації ключа та його під'єднання до графічного інтерфейсу, було протестовано роботу програмного додатку, результат зображено на рисунку 3.20.

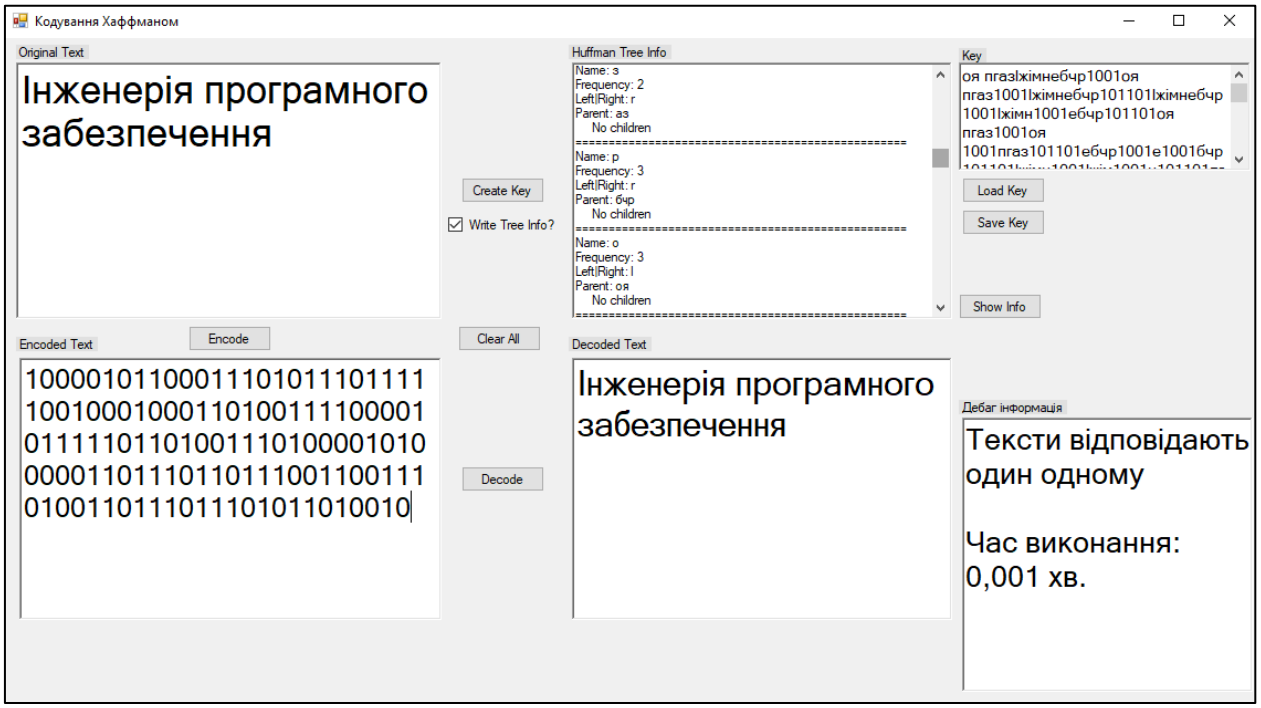


Рисунок 3.20 – Результат роботи додатку при генерації ключа для зашифрованого тексту

Після генерації ключа, необхідно розробити алгоритм зчитування та перетворення тексту у необхідну форму, для використання для подальшого дешифрування.

Алгоритм зчитування розбиває вхідний текст на рядки тексту у тексту, після чого використовується стандартна функція «String.Split()», що дозволяє

розділити один рядок на декілька різних, ділячи їх по певному роздільнику.

Після цього було застосовано перероблений код для генерації дерева Хаффмана із таблиці частот, для створення дерева Хаффмана за допомогою ключа. Після створення алгоритму та під'єднання його до графічного інтерфейсу, було перевірено правильність результату роботи додатку, де усі тести пройшли успішно.

На даному етані основний функціонал програмного додатку реалізований, але все ще є додаткові функції, які необхідно реалізувати. Ці функції є маленькою частиною усього проєкту, тому вони представляють собою одне завдання з оптимізації та доопрацюванні програмного додатку.

Було перероблено та розроблено «з нуля» такі функції додатку:

1. Додано

- функцію завантаження текстових файлів до програми, для їх подальшого використання;
- функцію збереження даних текстових вікон;
- реалізація умов перевірок вхідних даних для графічного інтерфейсу;
- фіксація розміру вікна додатку.

2. Перероблено:

- фінальна реалізація алгоритму шифрування та дешифрування та їх взаємодія між собою;
- метод виведення дерева Хаффмана та часу виконання;
- мова інтерфейсу повністю реалізована на англійській.

Після усіх цих змін, інтерфейс додатку набув іншого вигляду та функціоналу, який повністю покриває усі потенційні користувацьки потреби при використанні цього додатку. Рисунок з фінальним інтерфейсом основного вікна додатку та вікна додаткової інформації зображено на рисунку 3.21.

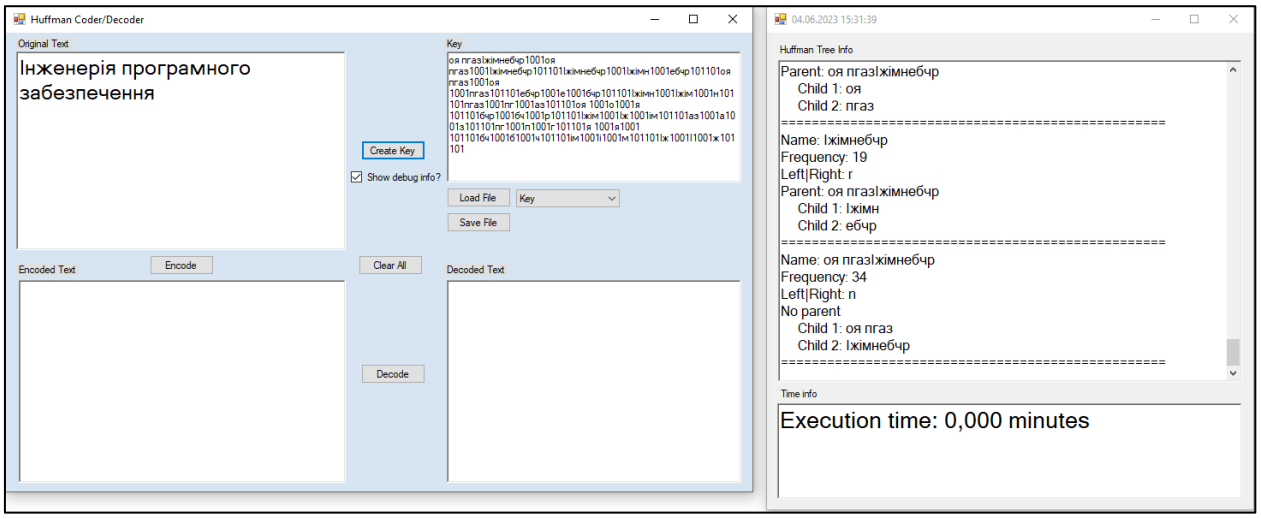


Рисунок 3.21 – Фінальний вигляд розробленого програмного додатку

Після завершення розробки програмного додатку, було проведено тестування за допомогою графічного інтерфейсу, де не було виявлено ніяких критичних помилок у алгоритму роботи додатку.

При оцінюванні потенційних, додаткових, змін, було виявлено можливість покращення графічного інтерфейсу додатку. У даному стані, графічний інтерфейс є дуже примітивним, хоча він виконує необхідні завдання зберігання даних. Також можливе додавання підтримки системи автоматичного шифрування та збереження даних за стандартним сценарієм, для полегшення роботи користувача з додатком при шифруванні великих обсягів окремих даних. Можливе додавання алгоритмів шифрування файлів різних типів, але це є окремою темою, яка потребує вивчання особливостей реалізації цих типів даних.

3.3 Використання системи контролю версій «Git»

Під час розробки програмного додатку, великою частиною його розробки була система контролю версій (VCS), яка є основою сьогоденного процесу розробки так само, як і технології та методології розробки програмного забезпечення.

Системи контролю версій дозволяють зберігати безліч різних версій програмного додатку на декількох пристроях одночасно та маючи можливість

комунікацій між цими пристроями для оновлення та зберігання загального стану розроблюваного проєкту. Можливість створення різних версій додатку дає змогу розробнику розроблювати потенційно небезпечні, для процесу розробки, зміни у проєкті, з можливістю відтворення початкового стану проєкту на декількох пристроях одночасно. Раніше, така система могла бути «саморобною реалізацією», що полягала у ручному створені копій проєктів на комп'ютері розробника, що є не самим ефективним і оптимізованим рішенням, бо уся «каталогізація» даних лежить на плечах розробника, через що помилки при розробці є неминучими. Саме цьому були створені системи контролю версій.

Використання системи контролю версій «Git» при розробці цього програмного додатку дозволило експериментувати з реалізацією додатку без можливості випадкового спотворення програмного коду без можливості його відтворення, з різних причин.

Система для зберігання даних, як було зазначено раніше, є сервісом «GitHub» який є безплатною системою для створення та збереження репозиторіїв з можливістю створення приватних та публічних репозиторіїв, що зробило цей сервіс дуже популярним та одним з основних на ринку цих послуг. Інтерфейсом, за допомогою якого здійснювалася робота з системою контролю версій, є звичайний термінал. На рисунку 3.22 зображено інформацію системи «Git», що надається при перевірці стану локального репозиторію за допомогою команди «git status».

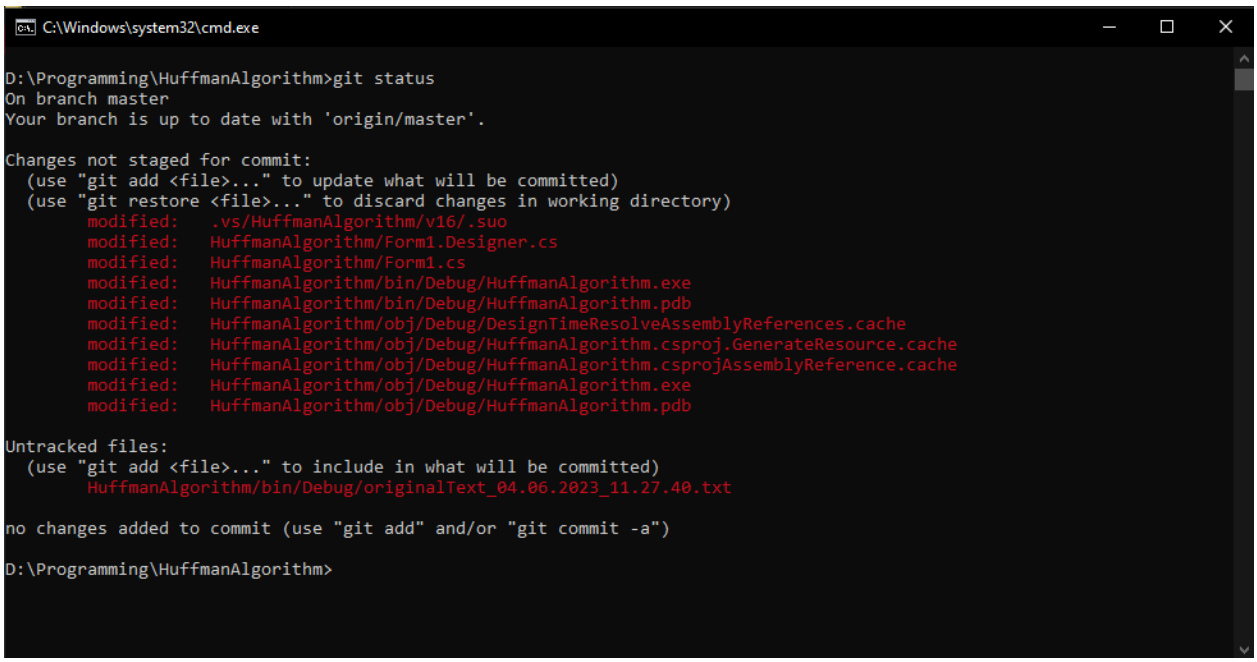


Рисунок 3.22 – Робота системи «Git» при перевірці локального статусу репозиторію

На рисунку 3.22 видно, що система контролю версій автоматично шукає усі зміни у локальній, обраній, гільці та пропонує додати його до «коміту».

«Коміт» (англ. «commit») – є елементом системи оновлення стану репозиторію, додаючи до нього зміни з локального або іншого центрального репозиторію. Для додавання усіх змін локального репозиторію до нового «коміту», необхідно ввести команду «git add -u», де атрибут «-u» вказує на уточнення команди «update», тобто «оновлення» усіх відстежуваних файлів, усі файли що не були, до цього, помічені як «відстежуємі файли», не будуть додані до «коміту». У самому списку файлів, що відстежуються можна побачити статус зміни файлу, у цьому проєкті, частіш за все, стани файлів можуть бути модифікованим чи видаленим, як зображено на рисунку 3.22. На рисунку 3.23 зображено командне вікно, після введення команд «git add -u» та «git status».

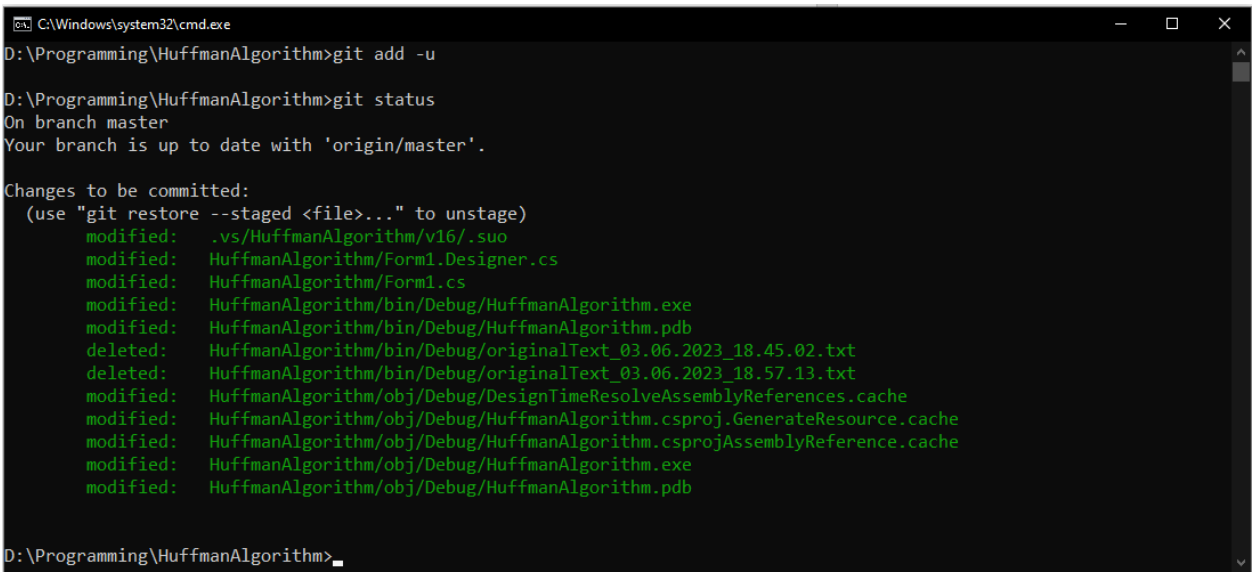


Рисунок 3.23 – Робота системи «Git» при додаванні файлів до «коміту»

На рисунку 3.23 можна побачити, що усі файли, що відстежувалися системою тепер мають зелений колір та знаходяться у списку «Changes to be commited» – «зміни які будуть закомічені». Якщо є необхідність видалення файлів з цього списку, можна використати команду «git restore»

Після додавання, оновлення чи видалення файлів з простору, який відстежується системою контролю версій, необхідно створити «коміт», для того, щоб його можна було завантажити до віддаленого репозиторію, який, у термінології «Git» називається «remote». Для відправлення змін у локальній версії проєкту до віддаленого репозиторію, необхідно створити «коміт» за допомогою команди «git commit -m “повідомлення яке буде описувати коміт”», для створення «коміту» та «git push назва_віддаленого_репозиторію назва_поточної_гілки». На рисунку 3.24 зображено командне вікно, після введення команд «git commit -m “final changes, deleted junk files”» та «git push origin master»

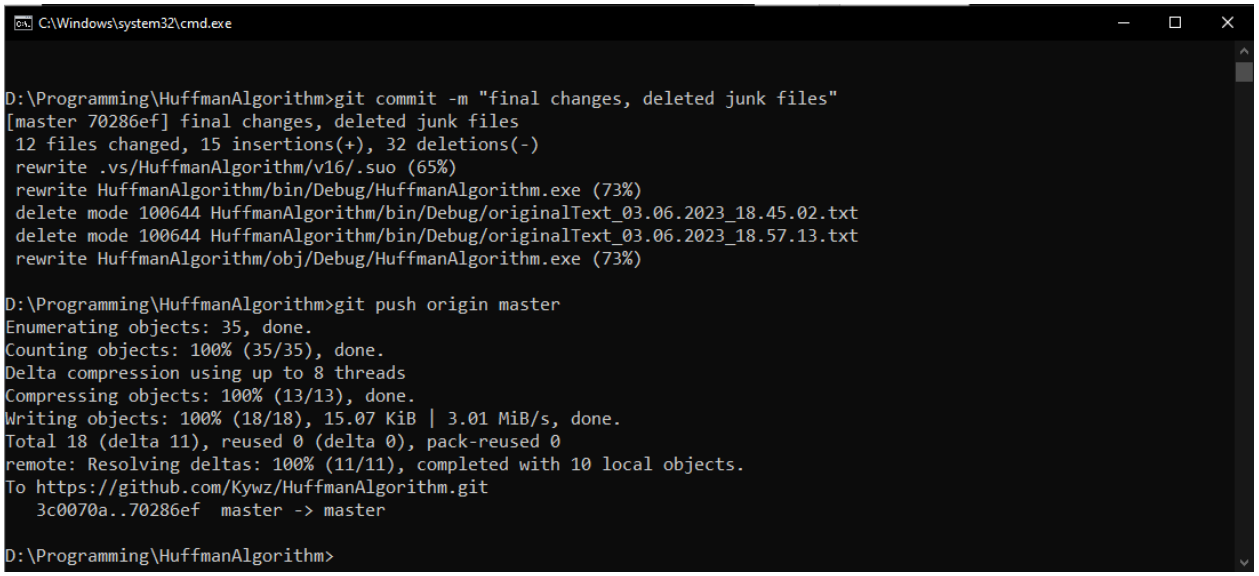


Рисунок 3.24 – Робота системи «Git» при створенні «коміту» та завантаження його до віддаленого репозиторію

Після завантаження усіх змін до віддаленого репозиторію – статус систему буде вказувати на повну актуальність даних віддаленого репозиторію з локальним.

У випадку змін файлів у віддаленому репозиторії – необхідно використовувати команду «git pull» або «git fetch», в залежності від ситуації.

При копіюванні проєкту без локального репозиторію, необхідно використовувати команду «git clone». яка створює копію віддаленого репозиторію на локальному ПК.

Основним та самим корисним елементом функціонала системи «Git» є система гілок та їх організації, які, як і було зазначено, дають змогу створювати копії файлів репозиторію та зберігати їх окремо від головної гілки, що дозволяє виконувати будь які зміни у гілках паралельно одне від одного. Для створення нової гілки необхідно використання команди «git branch назва_гілки», після використання якої буде створено нову гілку, на базі тієї, що обрана наразі за назвою, що була вказана. Усі зміни у створеній гілці не будуть ніяк відображатися у батьківській. Якщо з’являється необхідність у поєднанні цих гілок, можливе використання команди «git merge», яка поєднує дві гілки, при відсутності конфліктів між ними. При наявності конфліктів,

можливе їх усунення, обираючи варіант файлу, яких буде пріоритетним для нової, об'єднаної гілки. Об'єднання відбувається на базі обраної гілки, тобто інша гілка буде «закінчена» та її продовження бути йти по шляху першої гілки після об'єднання. Принцип роботи команди «git merge» зображено на рисунку 3.25, де гілка «master» на «коміті» «G» є поточною гілкою, а гілка «topic» є тією, що поєднується з «master».

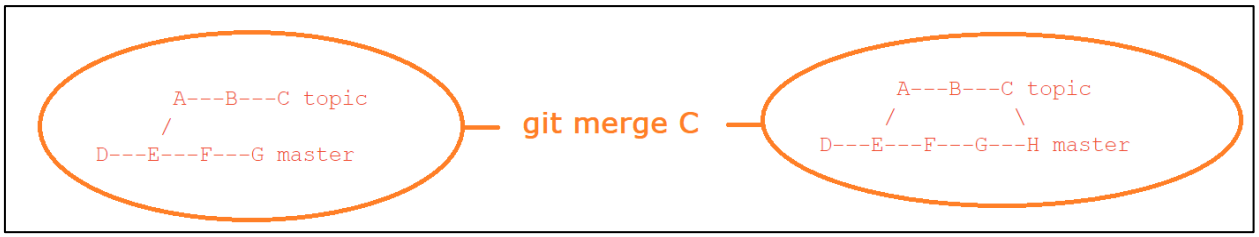


Рисунок 3.25 – Принцип роботи команди «git merge» системи «Git»

Усі зазначені команди були використані при розробці цього програмного додатку та роботі з системою «Git».

3.4 Тестування програмного додатку

Тестування програмних додатків є інтегральною частиною розробки будь-якого програмного додатку, але не завжди тестування проводять відповідні працівники сфери «Quality Assurance», які частіше називаються «QA».

«Наука QA» є дуже великою сферою світу IT, бо усі комерційні продукти та продукти критичного значення мають бути максимально працездатними, без можливостей критичних помилок під час їх роботи. Комерційні проєкти ризикують добутком, а проєкти що використовуються на критичних об'єктах, наприклад аеропорти, системи електропостачання та інші, повинні мати мінімальну кількість помилок перед тим, як вони почнуть використовуватися у їх відповідних сферах. Саме через це «QA» є останньою інстанцією у життєвому процесі будь-якого програмного забезпечення, бо саме цей відділ може позначити остаточну завершеність продукту.

Можна точно зазначити, що процес повного тестування програмного додатку програмістом, більше уповільнює процес розробки програмного продукту, аніж підвищує його, через необхідність моментального полагодження усіх недоліків додатку при його знаходженні, що може понести ще більше проблем. Також, частіш за все, ціна часу програміста значно вища за ціну часу тестера, через це вигідніше використання команд «QA» для перевірки роботи додатку.

Основні методології тестування програмних додатків зображено на рисунку 3.25.

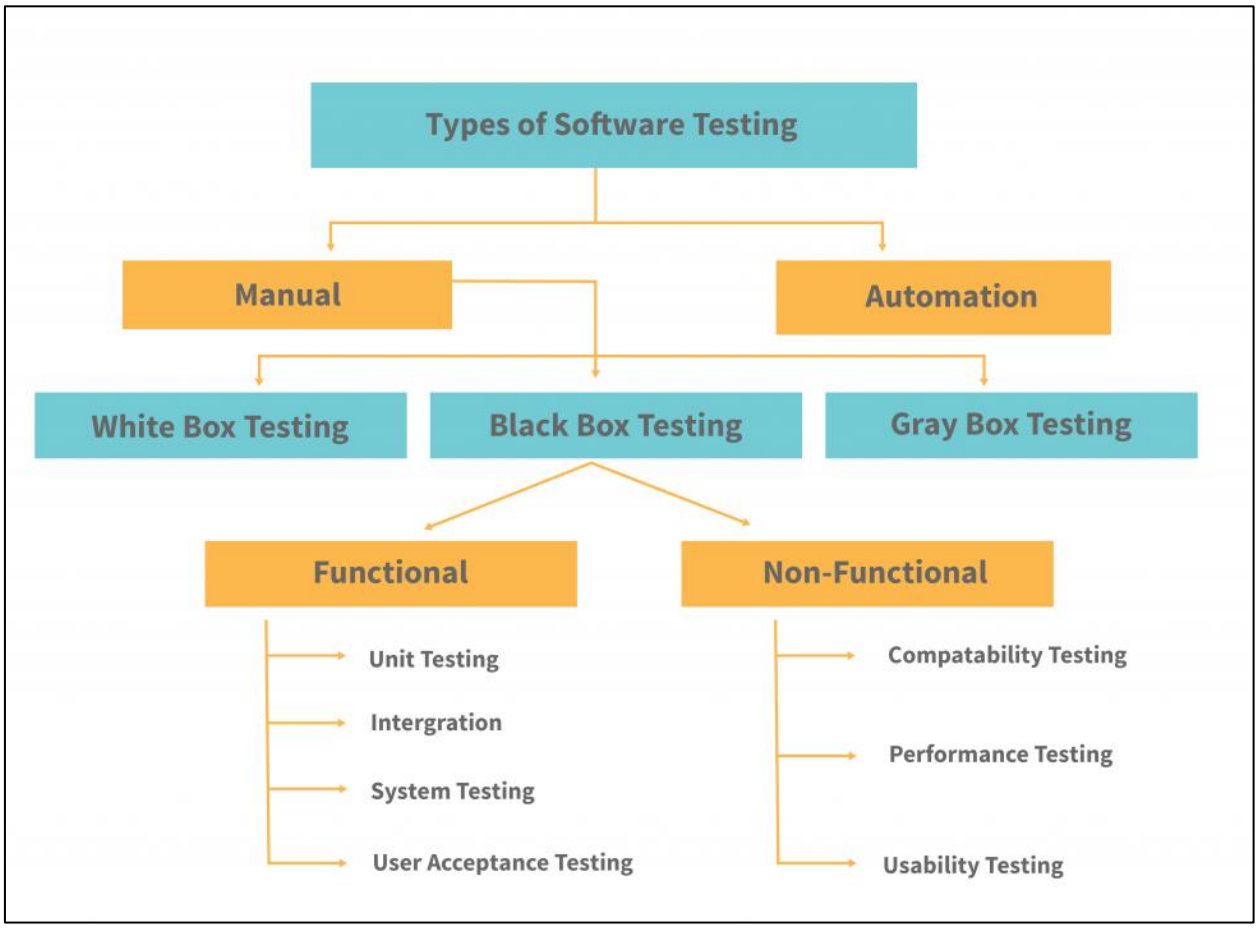


Рисунок 3.25 – Методи тестування програмного забезпечення

Під час розробки цього програмного додатку, було використано методи ручного тестування з методологіями «тестування білого ящика» та «чорного ящика».

Тестування «білого ящика» надає тестеру доступ до програмного коду. У такому випадку, тестер переглядає код та шукає будь-які помилки у логіці алгоритмів та коду. Цей метод не завжди використовується у розробці великих проєктів через велику кількість коду та не обов’язкове розуміння коду тестером, через різні рівні навчання та професіональності, але, на практиці, завжди використовується програмістом під час розробки. Ефективність цього методу не така велика, як при тестуванні «чорним ящиком», через можливі непередбачувані проблеми.

Тестування методом «чорного ящика» є самим розповсюдженим та самим технологічним, з боку кількості розроблених варіацій цієї методології. Цей метод включає у себе:

- unit testing (модульне тестування);
- integration testing (інтеграційне тестування);
- system testing (системне тестування);
- performance testing (тестування продуктивності);
- regression testing (регресійне тестування);
- інші методи, які не були використані під час розробки.

Модульне тестування – це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яка може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру. В об'єктно-орієнтованому програмуванні – метод.

Інтеграційне тестування – це фаза тестування програмного забезпечення, під час якої окремі модулі програми комбінуються та тестуються разом, у взаємодії. Інтеграційне тестування виконується після модульного тестування та перед верифікацією та валідацією ПЗ. Якщо розглядати цей процес як систему, то на вхід їй подаються модулі, які вже пройшли модульне тестування, потім модулі групуються в більші частини, виконуються тести передбачені планом, а на виході системи – інтегрована система, що готова до системного тестування.

Системне тестування є одним з рівнів тестування програмного забезпечення. Системне тестування тестує інтегровану систему для перевірки відповідності всім вимогам. Перевірка повноти та правильності документації користувача є важливою частиною системного тестування. Всі тестові комбінації повинні розроблятися тільки з використанням документації користувача.

Ці методи є, частіш за все, послідовними у своєму виконанні, де «unit testing», це тестування окремої функціональної одиниці у нейтральних умовах. Це тестування передбачує наявність однієї центральної, або основної, версії програмного додатку та багато інших функцій, які не інтегровані до центральної версії. Тестування цієї функціональної одиниці відбувається окремо від усіх інших одиниць. Після завершення етапу «unit testing», проходять «integration tests», де перевіряється робота функціональної одиниці з самою новою центральною версією, де нова функцій інтегрована до основної версії. Після цього відбувається «system tests», де перевіряється загальна робота додатку та взаємодію всіх його елементів між собою. Це є кінцевим етапом у тестування будь-якої функції, бо після цього, функція стає частиною проєкту. Але навіть після цього, існують регресійні тести, які передбачають перевірку роботи старих функцій при змінах, або при регулярних перевірках. Також, під час кожного етапу тестування, частіш за все, відбуваються тести ефективності роботи програмного додатку. Регресійні тести дозволяють підтримувати працездатність додатку при змінах архітектури та функціоналу додатку.

Усі зазначені етапи тестування є ключовими при розробці будь якої функціональної одиниці, але якщо проводити такий процес перевірки для кожної зміни у додатку, то проєкт ніколи не буде завершений, тому необхідно правильно описувати об’єм робот для кожної окремої функціональної одиниці, та, за можливістю, поєднувати їх, ця задача належить менеджерам проєктів, або «Project Manager», які займаються організацією загального процесу розробки програмного забезпечення.

Тестування програмних додатків також включає у себе методологію «Automation» - автоматизації, яка передбачає використання автоматизованих систем для тестування програмного додатку, використовуючи програмний код, замість ручного тестування. Цим, частіш за все, займаються окремі програмісти, основне завдання яких – розробка автоматизованих тестів. Але, на практиці, цим також займаються ті програмісти, які і створюють програмний код.

Також існує «тестування сірого ящика», яке є різними варіаціями методів «білого» та «чорного» ящика. Цей метод є «нестабільним» за своєю реалізацією та використанням, тому він варіюється від проєкта до проєкту, без певної конкретики.

Для збереження інформації про тести, існують структури, які називаються тест-кейси, які дозволяють організувати процес тестування у окремі елементи. Тест-кейси складаються з назви тест-кейсу, його очікуваного результату, кроків для його виконання та додаткових параметрів, які визначаються самою командою відділу тестерів. Тест-кейси, зазвичай, розроблюються для окремих функціональних одиниць, але можуть бути зібрані у тест-с'юти, які можуть бути використані як окремі тест-кейси, які є набором інших тест-кейсів, це додає можливість збору усіх окремих тест-кейсів для різних функцій у одну єдину систему, що дозволить використати її при тестуванні усієї системи додатку, що називається тест-планом. На рисунку 3.26 зображено структуру організації системи тест-кейсів.

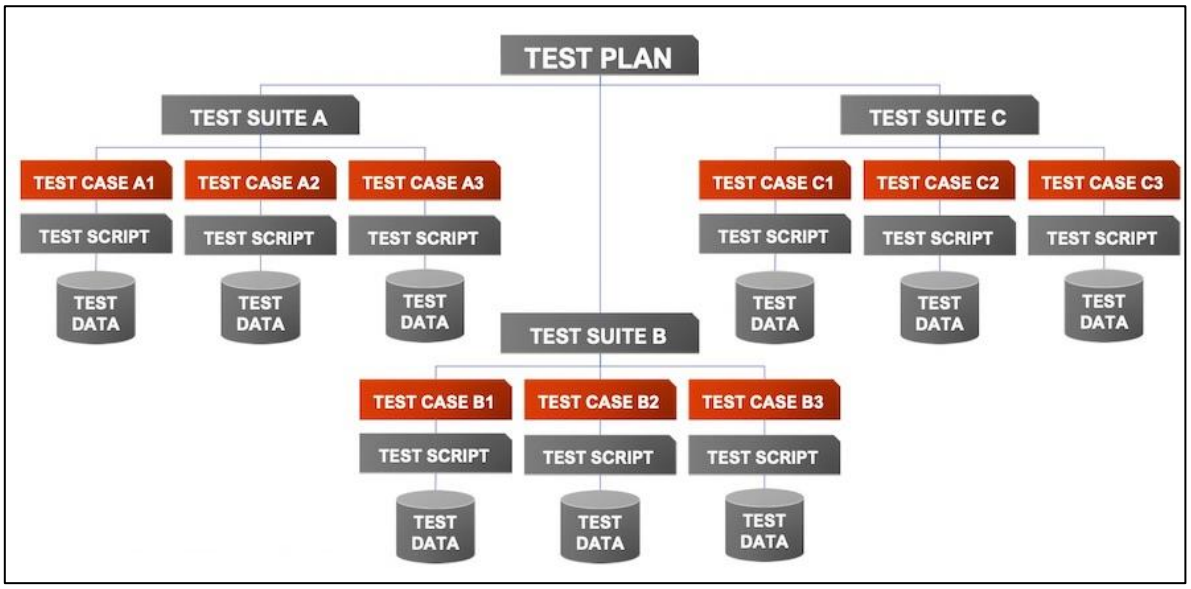


Рисунок 3.26 – Схема організації тест-кейсів

Тест-кейси, зазвичай, використовуються у великих проєктах під час розробки самої функціональної одиниці, для того, щоб під час початку тестування функції, тестер мав змогу оперативно та повністю протестувати усі необхідні елементи функції, та при зміні цієї функції або перенесені уваги на щось інше, не повинен був знов вивчати та оцінювати цю функціональну одиницю. Для маленьких проєктів тест-кейси не є доцільними, бо час на їх розробку значно більший, аніж проста перевірка усіх можливих станів додатку. Цей метод тестування називається «monkey testing», або «мавпяче тестування». Воно передбачає, що тестер не має ніякого досвіду роботи з функцію, або додатком, та ніякої супроводжуючої документації, через що тестер повинен самотужки вивчити принципи роботи програми, її особливості та стани. Цей метод є доволі поширеним при неорганізованій або маленькій компанії, де команда розробки не має певного досвіду, або не розроблює документацію до своїх продуктів. Цей метод тестування, у певній мірі, був застосований під час розробки цього додатку більш усього, так як він є ефективнішим зі сторони повного покриття додатку тестами, при розумінні усіх функцій додатку.

При тестуванні цього додатку, основними темами для тестів були тести на максимальне та мінімальне значення, де перевіряються максимальні та

мінімальне значення які можуть бути використанні під час використання цієї програми, тести на типи даних, які перевіряють правильність використання та розуміння даних додатком та тести на правильність алгоритмів додатку, що перевіряють роботу алгоритму Хаффмана у різних умовах.

Завдяки тестам розробленого програмного додатку, було виявлено велику кількість проблем між різними етапами реалізацію додатку, які були опрацьовані та ліквідовані. Система «Quality Assurance» виявилась дуже ефективним інструментом для перевірки додатку на несправності, навіть під час паралельної розробки програмного додатку.

3.5 Висновки за розділом

Під час реалізації програмного додатку, що реалізує усі необхідні функції для роботи алгоритму Хаффмана на мові «C#», було створено програмний додаток, що має графічний інтерфейс на базі фреймворку «.NET», що надає можливість використання цього додатку на різних системах без використання зовнішніх бібліотек. Було обрано інструменти для реалізації цього додатку, відносно поставлених умов та завдань. Було використано систему контролю версій «Git» та його реалізацію «GitHub» для організації розробки елементів програмного додатку. Було визначено основні методи проєктування та тестування програмного додатку. Було зазначено зміну ефективності роботи розробленого алгоритму шифрування при підвищенні розрахункових потужностей.

ВИСНОВКИ

Оптимізація даних є необхідним кроком у розвитку інформаційних технологій. Зі збільшення розрахункових потужностей, збільшується і кількість даних, що оброблюються, але розвиток технологій зберігання даних не є таким швидким, як розвиток усіх інших обчислювальних елементів розрахункових машин. Використання алгоритмів стиснення та оптимізації даних є ефективним компромісом для потреби у об’ємах пам’яті. Через швидкість обробки даних з’являється можливість реалізації більш складних та ресурсозатраних алгоритмів оптимізації, що дозволяє використовувати старі об’єми пам’яті зменшуючи об’єм пам’ять, що займається даними.

Розглянуто різні типи та реалізації алгоритмів, опрацьовано принцип їх роботи, плюси та мінуси. Опрацьовану систему оцінювання ефективності алгоритмів стиснення. Знайдено самий ефективний алгоритм для стиснення серед переглянутих. Розглянуто методики та принципи зберігання даних, типи та формати у яких ці данні перетворюються. Розглянуто практичне використання деяких з алгоритмів, їх застосування у сучасних технологіях. Розглянуто методи реалізації алгоритму Хаффмана за допомогою мови програмування «C#».

Реалізовано програмний додаток, здатний виконувати усі функції алгоритму Хаффмана, який має графічний інтерфейс з стандартними користувацькими функціями. Програмний додаток був перевірений на декількох пристроях та протестований використовуючи стандартні методології технології «Quality Assurance» та їх окремі методи. Було використано систему контролю версій та створено репозиторій за допомогою системи «GitHub», за допомогою якого було створено декілька версій додатку. Було створено блок-схеми роботи алгоритмів, що були реалізовані у додатку. Було набуто практичних навичок у розробці програмного забезпечення з використанням сучасних інструментів та технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Roughgarden T. Algorithms Illuminated Omnibus Edition. New York: Cambridge University Press, 2022. 690 p.

2. Salomon, D. A Concise Introduction to Data Compression. Berlin: Springer, 2008. 328 p.

3. Tank, M.K. "Implementation of Lempel-ZIV algorithm for lossless compression using VHDL". Implementation of Limpel-Ziv algorithm for lossless compression using VHDL. Thinkquest 2010: Proceedings of the First International Conference on Contours of Computing Technology. Berlin: Springer, 2011. 275–283 p.

4. Scully D., Brodley C.E. "Compression and machine learning: A new perspective on feature space vectors". Data Compression Conference, 2006. 332 p.

5. Schroeder, Manfred R. "Bell Laboratories". Acoustics, Information, and Communication: Memorial Volume in Honor of Manfred R. Schroeder. Berlin: Springer, 2014. 480 p.

6. Luo, Fa-Long. Mobile Multimedia Broadcasting Standards: Technology and Practice. Berlin: Springer Science & Business Media, 2008. 685 p.

7. Luenberger D.G. Linear and Nonlinear Programming/ D.G. Luenberger, Y. Ye. – Heidelberg: Springer, 2016. 546 p.

8. Carter M.W., Price C.C., Rabadi G. Operations research: a practical approach Boca Raton: CRC Press, 2019. 471 p.

9. Nagel C. Professional C# And .NET. Wiley & Sons, Incorporated, John, 2021. 1008 p.

10. Loeliger J. Version control with Git. 2nd ed. Beijing: O'Reilly, 2012. 434 p.

11. M. McQuaid. Git in Practice. Shelter Island, NY, USA: Manning Publications, 2015. 249 p.

12. Skeet J., Lelek T. Software Mistakes and Tradeoffs: How to Make Good Programming Decisions. Manning Publications Co. LLC, 2022. 416 p.

13. Skeet J. C# in Depth. Manning Publications Co. LLC, 2019. 528 p.

14. Troelsen A., Japikse P. Pro C# 7: With .NET and .NET Core. Apress, 2017. 1372 p.

15. Seebeck A. C# Fundamentals - Getting Started with C# 11 And .NET 7. unQbd, 2022. 168 p.

16. Khorikov V. Unit Testing Principles, Practices, and Patterns. Manning Publications Company, 2020. 304 p.

17. Buonanno E. Functional Programming in C#, Second Edition. Manning Publications Co. LLC, 2021. 448 p.

18. Troelsen A., Japikse P. Pro C# 10 With .NET 6: Foundational Principles and Practices in Programming. Apress L. P., 2022. 1705 p.

19. Price M. J. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development: Build applications with C#, .NET Core, Entity Framework Core, ASP.NET Core, and ML.NET using Visual Studio Code, 4th Edition. Packt Publishing, 2019. 818 p.

20. S. Chacon. Pro Git, Second Edition, Kindle Edition. Berlin: Springer, 2014. 458 p.

Додаток А

Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження
С. 6-8,51		Невірні переліки
С. 9		Змінні в формулі та тексті повинні бути однаковими
ПЗ		Нещільне заповнення сторінок ПЗ
Зміст		Нема додатків

Додаток Б

Фрагмент лістингу додатку

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
// github: https://github.com/Kywz/HuffmanAlgorithm
// This method contains main algorithms for huffman tree operations
namespace HuffmanAlgorithm.HuffmanCode
{
    class HuffmanTree
    {
        public static List<HuffmanElement> createHuffmanTree(List<String> key)
        {
            List<HuffmanElement> huffmanTree = new List<HuffmanElement>();
            string[] words;
            string[] splitWord = { "1001" };
            HuffmanElement elementParent;
            HuffmanElement elementLeft;
            HuffmanElement elementRight;
            foreach (string item in key)
            {
                words = item.Split(splitWord, StringSplitOptions.RemoveEmptyEntries);
                words[2] = words[2].Replace("101101", "");
                if (huffmanTree.Count != 0) {
                    elementLeft = new HuffmanElement(words[1]);
                    elementLeft.leftRight = 'l';
                    elementLeft.parent = huffmanTree.Find(instance =>
instance.name.Equals(words[0]));
                    elementRight = new HuffmanElement(words[2]);
                    elementRight.leftRight = 'r';
                    elementRight.parent = huffmanTree.Find(instance =>
instance.name.Equals(words[0]));
                    huffmanTree.Find(instance =>
instance.name.Equals(words[0])).child.Add(elementLeft);
                    huffmanTree.Find(instance =>
instance.name.Equals(words[0])).child.Add(elementRight);
                    huffmanTree.Add(elementLeft);
                    huffmanTree.Add(elementRight);
                }
                else
                {
                    elementParent = new HuffmanElement(words[0]);
                    elementLeft = new HuffmanElement(words[1]);
                    elementLeft.leftRight = 'l';
                    elementLeft.parent = elementParent;
                    elementRight = new HuffmanElement(words[2]);
                    elementRight.leftRight = 'r';
                    elementRight.parent = elementParent;
                    elementParent.child.Add(elementLeft);
                    elementParent.child.Add(elementRight);
                    huffmanTree.Add(elementParent);
                    huffmanTree.Add(elementLeft);
                    huffmanTree.Add(elementRight);
                }
            }

            return huffmanTree;
        }
        public static List<String> createHuffmanKey(List<HuffmanElement> inputList)
        {
            //end of word symbol 1001
            //left child = 0; right child = 1;
```

```
            string splitWord = "1001";
            string endWord = "101101";
            List<String> huffmanKey = new List<String>();
            foreach (HuffmanElement huffElem in inputList)
            {
                if (huffElem.child.Count() == 2)
                {
                    huffmanKey.Add(huffElem.name + splitWord +
huffElem.child.First().name + splitWord + huffElem.child.Last().name + endWord);
                }
            }
            huffmanKey.Reverse(); //Reversing key for more tree-like structure
            return huffmanKey;
        }
        public static List<HuffmanElement> getHuffmanTree(List<HuffmanElement> inputList)
        {
            HuffmanElement newElementBuffer;
            List<HuffmanElement> bufferList = inputList;
            while (inputList.Count() != 1)
            {
                // Sorting list by ascending for proper work of Huffman algorithm
                inputList = inputList.OrderBy(instance => instance.frequency).ToList();
                // Creating new element from two elements
                newElementBuffer = new HuffmanElement(inputList.ElementAt(0).name +
inputList.ElementAt(1).name);
                newElementBuffer.frequency = inputList.ElementAt(0).frequency +
inputList.ElementAt(1).frequency;
                // Changing child#0 internal vars in the output list
                bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(0).name)).parent = newElementBuffer;
                bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(0).name)).leftRight = 'l';
                // Changing child#1 internal vars in the output list
                bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(1).name)).parent = newElementBuffer;
                bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(1).name)).leftRight = 'r';
                // Adding created element to the output list
                bufferList.Add(newElementBuffer);
                // Adding children to created element
                bufferList.Last().child.Add(bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(0).name)));
                bufferList.Last().child.Add(bufferList.Find(instance =>
instance.name.Equals(inputList.ElementAt(1).name)));
                // Preparation for next iteration
                inputList.RemoveRange(0, 2);
                inputList.Add(newElementBuffer);
            }

            return bufferList;
        }
        public static BitArray Encode(string source, List<HuffmanElement> encodingTree)
        {
            string buffString;
            List<bool> encodedSource = new List<bool>();
            foreach (char c in source)
            {
                buffString = encodingTree.Find(instance =>
instance.name.Equals(c.ToString())).getEncoding();
                buffString = reverseString(buffString);
                foreach (char cs in buffString)
                {
                    if (cs.Equals('0'))
                    {
                        encodedSource.Add(false);
```

```
                    }
                    else if (cs.Equals('1'))
                    {
                        encodedSource.Add(true);
                    }
                }
            }
            BitArray bits = new BitArray(encodedSource.ToArray());
            return bits;
        }
        private static string reverseString(string s)
        {
            char[] charArray = s.ToCharArray();
            Array.Reverse(charArray);
            return new string(charArray);
        }
        public static string Decode(BitArray bits, List<HuffmanElement> HuffmanTree)
        {
            HuffmanTree = HuffmanTree.OrderBy(instance => instance.name.Length).ToList();
// Sorting tree by name length, for no further use of frequencies
            string output = "";
            HuffmanElement buffElement = HuffmanTree.Last();
            foreach (bool b in bits) // Iterator for looking through encoded text
            {
                if (b == false)
                {
                    buffElement = buffElement.child.ElementAt(0); // Getting left child
from parent
                }
                else if (b == true)
                {
                    buffElement = buffElement.child.ElementAt(1); // Geting right child
from parent
                }
                if (buffElement.name.Length == 1) // Got to end of tree, length of name
== 1 says that there is no next element possible
                {
                    output += buffElement.name;
                    buffElement = HuffmanTree.Last();
                    continue;
                }
            }
            return output;
        }
    }
}
```

Додаток В

Матеріали презентації

МІНІСТРЕСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

Презентація до дипломної роботи
за темою: **«Алгоритми оптимізації та стиснення даних»**
зі спецчастиною: **«Розробка та реалізація алгоритму Хаффмана з застосуванням C#»**

Виконав: ст. гр. ІПЗ-19 Данило Стреляєв
Керівник: к.т.н., доц. Наталія Маслова

АКТУАЛЬНІСТЬ РОЗРОБКИ

Актуальність розробленого програмного додатку полягає у зменшенні об’ємів пам’яті, що займаються даними. У сучасному світі кількість об’ємів даних постійно зростає, через розвиток комерційних технологій використання даних користувачів та технологій штучного інтелекту, що потребують великого об’єму пам’яті для зберігання даних.

2

МЕТА ТА ПРАКТИЧНА ЧАСТИНА РОБОТИ, ОБ'ЄКТ ДОСЛІДЖЕННЯ

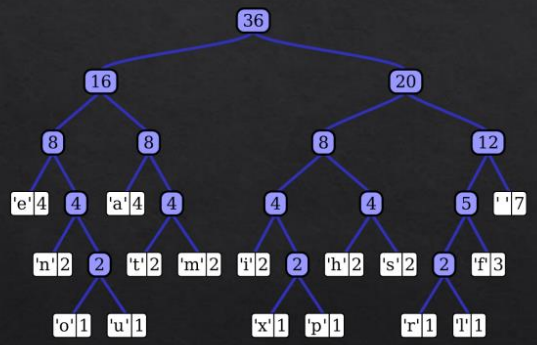
- **Об'єкт дослідження**: технології стиснення та оптимізації даних.
- **Мета роботи**: розробка програмного додатку, у якому реалізована робота алгоритму шифрування та дешифрування Хаффмана, з використанням лінійного методу зберігання ключів дешифрування.
- **Практична частина**: створенні алгоритму, за допомогою якого можливе ефективне стиснення даних та можливість декодування зашифрованих даних зі стовідсотковою ймовірністю, без втрати даних.

3

ЕФЕКТИВНІСТЬ МЕТОДУ, ВІДНОСНО АЛЬТЕРНАТИВ

4

ПРИНЦИП РОБОТИ АЛГОРИТМУ. ШИФРУВАННЯ



Основна ідея алгоритму Хаффмана полягає у використанні дерев Хаффмана, яка будується за допомогою таблиці частот, де елементи з найбільшою частотою отримують найменшу, за довжиною, бінарну послідовність, через що загальна довжина вхідного тексту матиме менший бітовий об'єм, аніж у початковому кодуванні (ASCII або UTF)

5

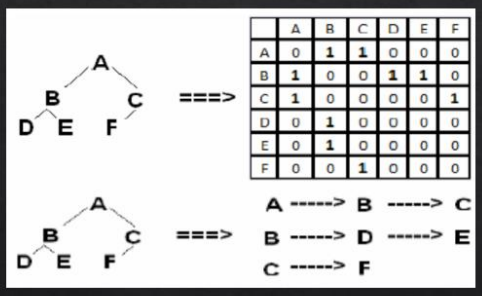
ПРИНЦИП РОБОТИ АЛГОРИТМУ. ДЕШИФРУВАННЯ

Дешифрування відбувається при послідовному проходженні по елементам закодованого тексту та бінарному дереву Хаффмана, за допомогою якого відбувається пошук закодованого значення, яке є елементом, що не має після себе «дітей».

З зазначеного алгоритму можна визначити, що текст не може бути дешифрований схожим або іншим деревом, через що цей метод також надає елемент захисту даних. Дешифрування іншими методами може зайняти дуже багато часу.

6

ПРИНЦИП РОБОТИ АЛГОРИТМУ. КЛЮЧІ



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	0	1	1	0
C	1	0	0	0	0	1
D	0	1	0	0	0	0
E	0	1	0	0	0	0
F	0	0	1	0	0	0

Необхідним елементом для шифрування та дешифрування є ключ.

Для дерева Хаффмана існує декілька варіантів реалізації, наприклад **графічна** – дерево Хаффмана, **таблична** – таблиця суміжності, та **лінійна**, яка використана у цьому проєкті.

Програмна реалізація не може ефективно використовувати графічний метод, таблична є не такою оптимальною та простою у реалізації, як лінійна.

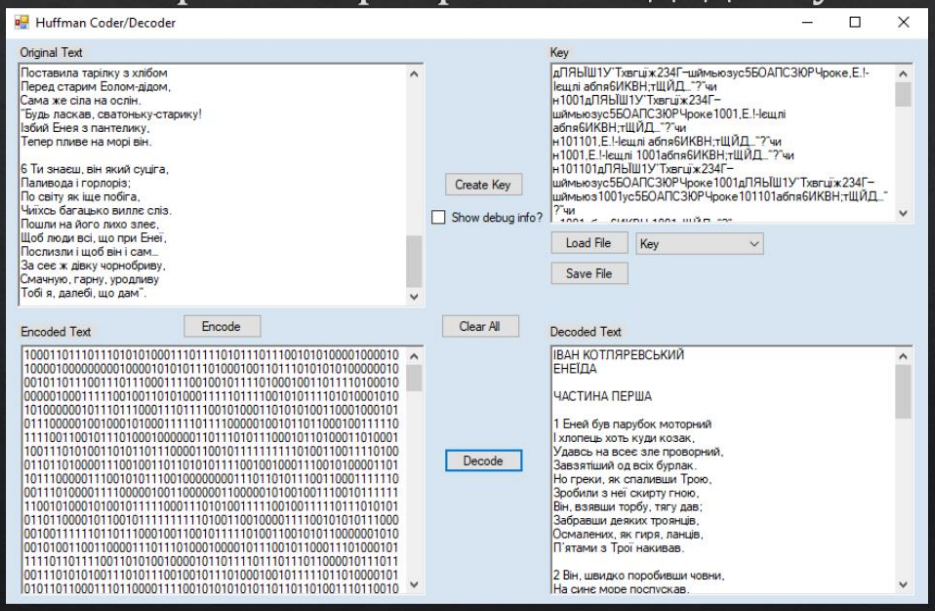
7

Інструменти для проєктування

- Графічний редактор «Draw.io».
- Середа розробки «Microsoft Visual Studio».
- Система контролю версій «GitHub».

8

Розробка програмного додатку



Використання системи Git

За час розробки, було створено декілька гілок та комітів, які у кінці розробки були зібрані у фінальну версію.

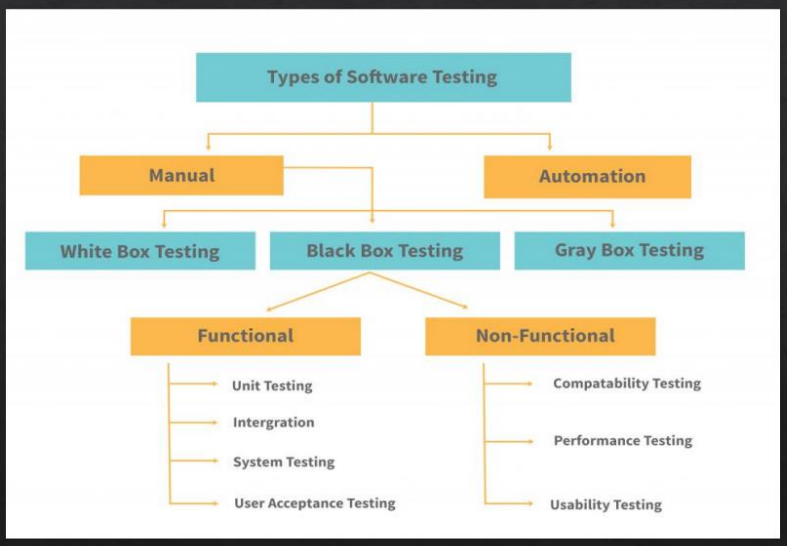
ВИСНОВКИ

В результаті виконання роботи:

- Розглянуто різні типи та реалізації алгоритмів **оптимізації та стиснення даних.**
- Опрацьовано принцип їх роботи, переваги та недоліки.
- Реалізовано програмний додаток, здатний виконувати усі функції алгоритму Хаффмана, який має графічний інтерфейс з стандартними користувацькими функціями.
- Проведено тестування та перевірка додатку на декількох пристроях з застосуванням методологій «Quality Assurance» та їх окремих методів.

12

Тестування розробки



11


Дякую за увагу!